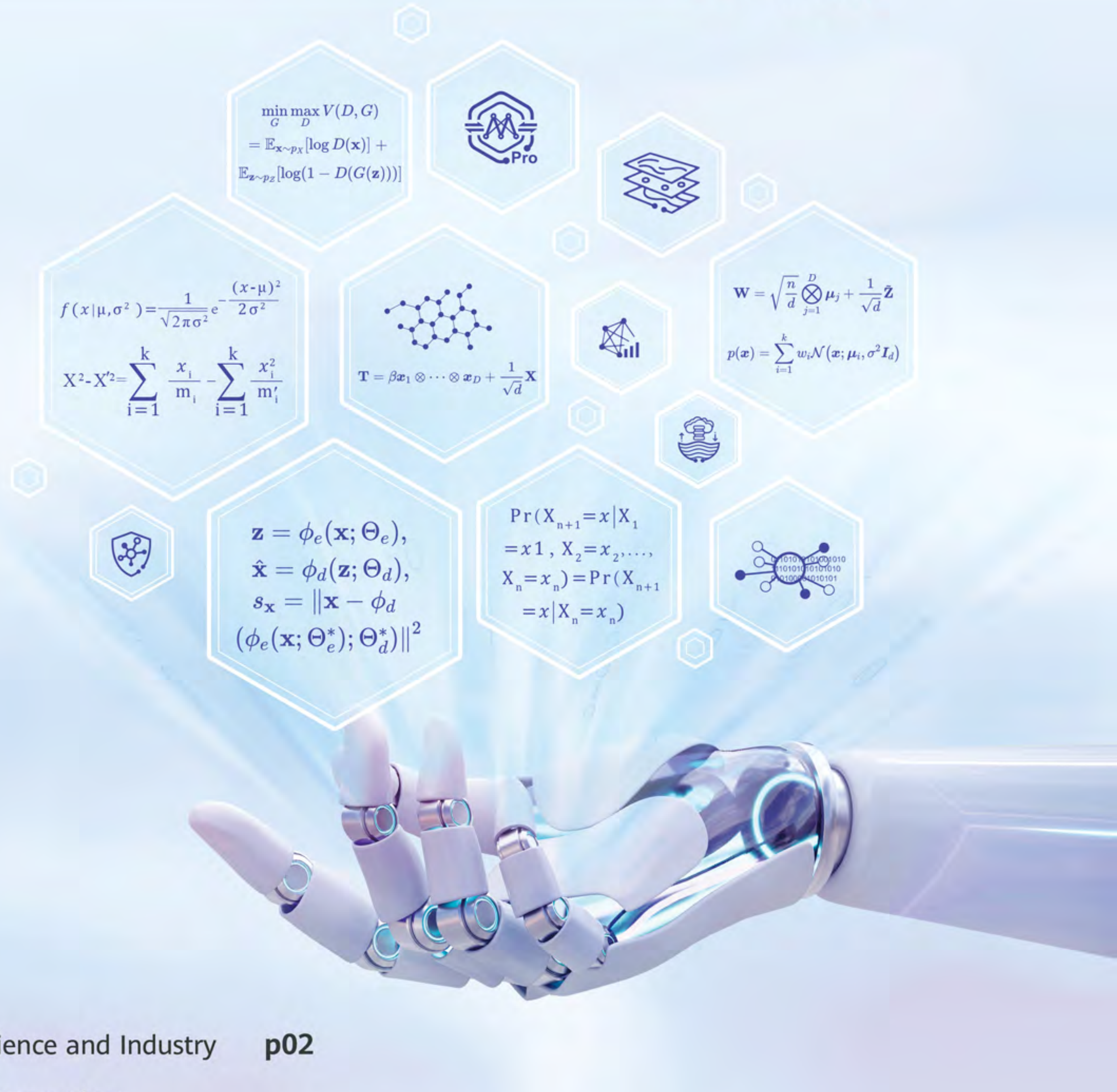
Communications of HUAWEI RESEARCH

June 2024
Issue 6


$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_X}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_Z}[\log(1-D(G(\mathbf{z})))]$
Pro
$f(x|\mu,\sigma^2)=\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
$X^2-X'^2=\sum_{i=1}^{k}\frac{x_i}{m_i}-\sum_{i=1}^{k}\frac{x_i^2}{m_i'}$
$\mathbf{T}=\beta\boldsymbol{x}_1\otimes\cdots\otimes\boldsymbol{x}_D+\frac{1}{\sqrt{d}}\mathbf{X}$
$\mathbf{W}=\sqrt{\frac{n}{d}}\bigotimes_{j=1}^{D}\boldsymbol{\mu}_j+\frac{1}{\sqrt{d}}\tilde{\mathbf{Z}}$
$p(\boldsymbol{x})=\sum_{i=1}^{k}w_i\mathcal{N}(\boldsymbol{x};\boldsymbol{\mu}_i,\sigma^2\boldsymbol{I}_d)$
$\mathbf{z}=\phi_e(\mathbf{x};\Theta_e),$
$\hat{\mathbf{x}}=\phi_d(\mathbf{z};\Theta_d),$
$s_{\mathbf{x}}=\|\mathbf{x}-\phi_d(\phi_e(\mathbf{x};\Theta_e^*);\Theta_d^*)\|^2$
$\Pr(X_{n+1}=x|X_1=x1, X_2=x_2,\ldots,X_n=x_n)=\Pr(X_{n+1}=x|X_n=x_n)$

HUAWEI

Editorial Note

In this *Huawei Research* issue, we explore AI's transformative power in industrial and scientific modeling and computing. Our focus encompasses a broad spectrum of topics, including computing architectures, data types, the intricacies of hardware-software co-design, the development of algorithm models and architectures, and the exploration of various theories. Our objective is to offer an in-depth yet easily understandable analysis of the prevailing challenges and opportunities in AI technologies.

We systematically explain how AI can be effectively applied in industrial and scientific modeling and computing to address problems that have remained unsolved for over 200 years and are challenging to solve using existing AI statistical modeling. We explore how computer graphics and multimedia technologies can be integrated to create holographic media representations and non-geometric 3D scenario modeling. Additionally, we discuss the development of intelligent devices that can work in harmony with humans and complex environments. We also delve into the development of robotics without the use of coordinate systems. Finally, we examine how nonlinear system signals can be processed in the signal and system domain, leading to more accurate modeling methods. We aim to provide valuable insights and guidance for the next generation of AI technologies to tackle complex challenges.

Spatial Computing introduces a novel, highly scalable architecture that transcends the conventional von Neumann model, designed to cater to the exponential computing power demands essential for AI advancements. *Reprioritizing Speculative Task-Level Parallelism* presents Hive, a cutting-edge task-based execution model and multicore architecture that enhances performance and optimizes energy efficiency. Hive leverages a wealth of fine-grained parallelism inherent in algorithms, employing dynamic priority updates to optimize execution. Hive ensures the integrity of speculative scheduling updates and prevents spurious task conflicts, establishing itself as an industry-leading hardware solution that significantly outperforms software-only parallel schedulers in efficiency and performance. *Exploration of Hybrid Optical-Electrical Switching Networks in AI Training Clusters* presents a groundbreaking hybrid optical-electrical switching network tailored for large-scale, high-bandwidth, and adaptable operations, addressing the challenges of cost and power consumption prevalent in computationally intensive scenarios such as AI and high-performance computing (HPC). Additionally, it introduces a novel collective communication algorithm optimized for this hybrid network, enhancing the efficiency of communication operations within AI training clusters. *Ascend HiFloat8 AI Training and Inference* doubles the computing power with a minimal increase in area by introducing an innovative 8-bit floating-point format, HiF8. This development, coupled with HiF8-based AI training and inference solutions, marks a substantial improvement in computing efficiency. To overcome hardware performance bottlenecks caused by complex CPU instructions for floating-point square root (FP SQRT) computation, *DP SQRT Computation Principle and Ultra-Low Latency Microarchitecture Design* introduces an FP SQRT computation precision doubling method. This method segregates high and low bits and is paired with a corresponding microarchitecture design, enhancing computing precision and system performance.

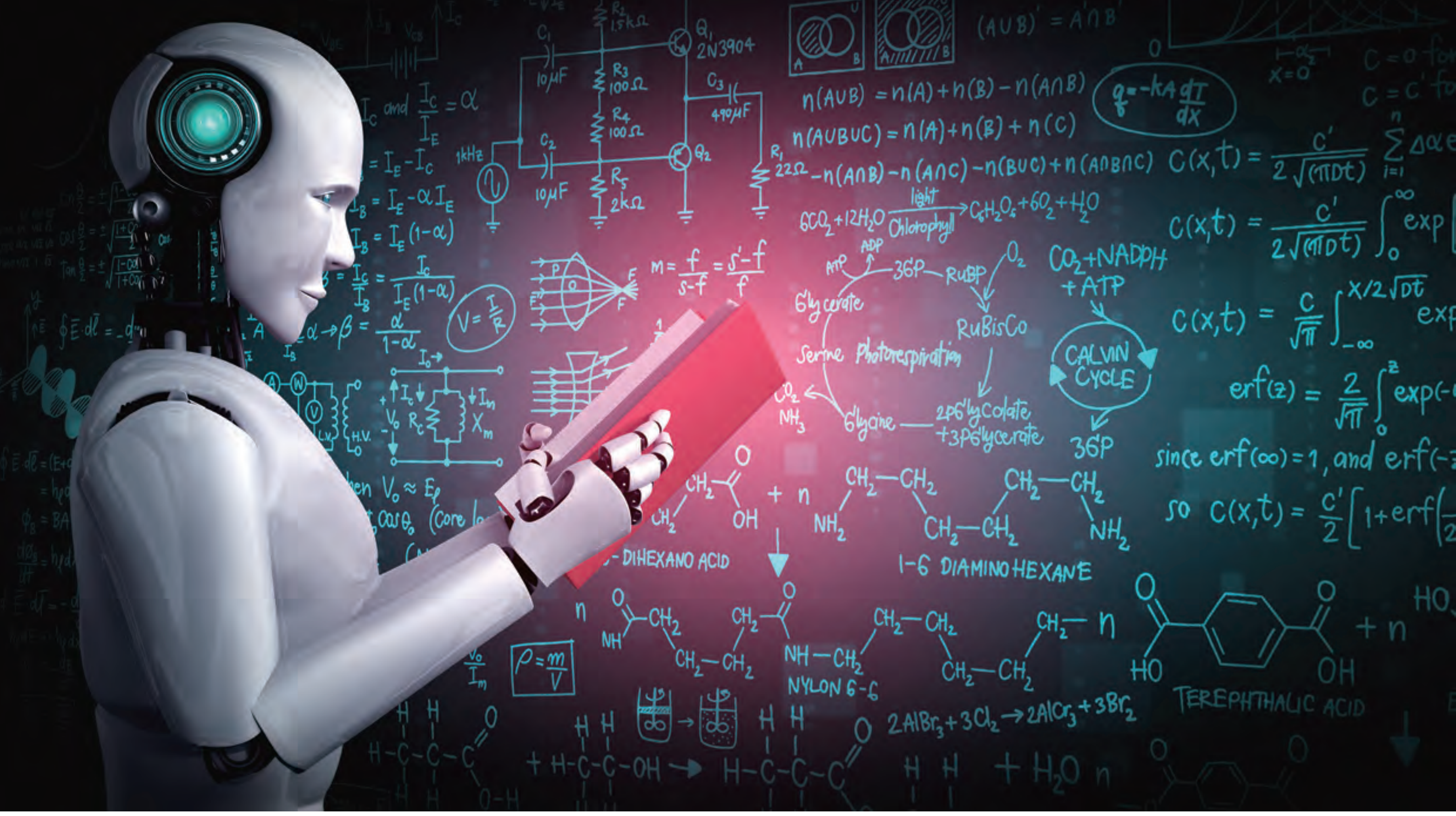
In hardware-software co-design and optimization, modern language implementations are increasingly leveraging dynamic or just-in-time (JIT) compilation techniques. This approach capitalizes on a unique opportunity to monitor and analyze the state of a program during its execution, allowing for real-time optimizations and enhancements that traditional compilation methods cannot offer. *Balancing the Yin and Yang of Dynamic Compilation and Execution* proposes innovative strategies for hardware-software co-design. It introduces software-directed narrowly focused profiling on hardware to more detailed profiles with minimal overhead. *Data-Centric Auto-Tuning of High-Performance Computing Applications* presents DCTuner, a novel auto-tuning method for optimizing HPC applications. DCTuner uses a data-centric representation, integrating a pruning strategy and a greedy exploration algorithm. This method enhances the performance of HPC applications beyond the current state-of-the-art benchmarks while ensuring high portability. *Humble Heroes* proposes a novel Algebraic Programming (ALP)/Pregel paradigm that scales well on a shared-memory parallel system and achieves speedups of up to 17.8x on common graph workloads, demonstrating that multiple humble programming models can be supported by a single software stack. *Asynchronous Training and MoRe* introduces the Momentum Reconstruction (MoRe) optimization technique, which achieves state-of-the-art convergence rates and generalization properties while reducing their memory requirements by a factor of two. *Computational Graph Representation of Equations System Constructors in Hierarchical Circuit Simulation* suggests using a computational graph representation to create an equations system constructor that supports dynamic parameters. This method, along with its JSON format netlist, simplifies the model development process and makes it easier to compute the gradients of equation remainders with respect to parameters.

In multidomain multimodal AI algorithms, *MDMMT-2* delves into the analysis of datasets and training methodologies for text-image and text-video retrieval tasks. It introduces a sophisticated multistage training strategy for multilingual models, enhancing knowledge transfer efficiency and facilitating the use of noisy datasets during training without compromising prior knowledge integrity. *Wasserstein Robust Reinforcement Learning* proposes an innovative zero-order optimization method to address the max-min game in reinforcement learning, incorporating a constraint based on Wasserstein distance. This method demonstrates superior performance and efficiency in low- and high-dimensional MuJoCo environments. *Random Tensor Theory, Algorithms and Applications* comprehensively explores tensor properties and classical tensor decomposition methods. It examines the application of random tensor theory in evaluating the performance of supervised and unsupervised learning, highlighting the pivotal role of tensors in AI applications. *Dynamical Systems and Control Theory Perspective of Computation* presents a groundbreaking perspective, conceptualizing computation as a control system that transitions from an initial state to a desired output. This novel approach integrates dynamical systems and control theory. It offers a framework for designing hybrid physical systems that balance energy consumption and accuracy, paving the way for substantial energy savings in large-scale computational tasks.

Heng Liao

Heng Liao
Chief Scientist of 2012 Labs

CONTENTS



AI for Science and Industry

Heng Liao, Linfeng Zhang, Lin Li

Abstract

In practical terms, connectionist AI algorithms have — over the past decade — functioned as a more effective statistical modeling approach for non-physical modeling tasks such as image classification, image segmentation, and machine translation. The development of connectionist AI has given rise to many new industries and brought huge economic benefits to them and society as a whole. AI algorithms have quickly drawn wide attention from both academia and industry thanks to their ability to solve problems related to the aforementioned modeling tasks — such problems could not otherwise be solved by using physical (or rule-based) modeling methods, which had been a common practice over the past two centuries of scientific development. Throughout that time, though, many difficult yet important problems remain unsolved or have not been adequately addressed. Some of these problems cannot be solved using only traditional methods, and even directly applying AI-based statistical modeling methods fails to produce any practical outcome for most of them. Worse yet is the wide variety of applications in which these problems are involved: scientific and industrial problem modeling and solving; integration of computer graphics and multimedia technologies (e.g., building holographic multimedia representations and modeling 3D scenarios with a non-geometric approach); manufacturing intelligent devices that collaborate harmoniously with humans and complex environments, or developing robotics without using coordinate systems; using highly structured advanced knowledge systems to realize automatic Q&A for both open and closed domains, and automated theorem proving; and processing nonlinear system signals in the signal and system domain, and developing higher-accuracy modeling methods for doing so. In this article, we review existing research for these applications and provide further insights.

Keywords

high-performance computing, AI

1 Introduction

AI algorithms have created incredible economic value over the past decade thanks to their increasing applications in several key fields.

In the security protection field, AI algorithms have made significant strides in computer graphics — they are comparable to or even outperform humans in image classification, segmentation, compression, enhancement, and recognition. And thanks to improvements in massive data storage, reading, and retrieval, AI algorithms have had a major impact in not only facilitating but also enhancing security protection across society.

AI algorithms have also been instrumental in the field of mobile phone applications. Take the camera snapshot function of mobile phones as an example — AI-based image processing helps to effectively ensure the quality of photo snapshots.

Another field in which AI algorithms create significant value is the autonomous driving industry. Here, AI technologies are heavily utilized in modeling the environment of vehicles and road obstacles, as well as in performing sensing, prediction, decision-making, control, and battery management functions.

And in the Internet field, AI algorithms enable big data analytics to do a remarkably good job in user profiling, thereby allowing content to be more personalized in order to achieve a better Internet surfing experience for users.

Essentially, AI algorithms benefit immensely from a data-driven neural network architecture of deep learning and reinforcement learning where these algorithms are rooted. Unlike the analytical approach adopted by traditional computer algorithms, AI algorithms leverage big data statistics to solve problems in a much more effective way — this is even more noticeable when addressing certain types of problems that the traditional approach cannot solve. Nevertheless, the huge success of applying AI algorithms in the fields mentioned earlier also leads to a high homogeneity of research. So while researchers commit valuable resources to these key fields, they might inadvertently overlook other more important and difficult areas that require even larger investment.

This article focuses on several important areas where established science or existing AI technologies fail to fully solve certain problems. It will explore the following topics:

- A New Approach to Scientific Computing and Industrial Computing
- A Novel Method of Integrating Computer Graphics and Multimedia Technologies
- A Groundbreaking Paradigm of Developing Autonomous Robotics
- A Fresh Understanding on the Learning and Reasoning of Structured Knowledge
- Material Basis of AI

2 A New Approach to Scientific Computing and Industrial Computing

Experiments, theories, and computing have been and continue to be three of the most important factors in advancing human science, enabling us to obtain and expand our control over information, energy, and matter. Before big data analytics emerged, processing information reliably depended primarily on physical modeling that utilizes mathematical symbols. Physical models have been advantageous because they are simple and specific, allow deductive reasoning, partially support analytical and numerical computation, and — very importantly — some of them are quantitatively verifiable through experiments. For instance, once Newton defined the concepts of point mass, rigid body, and the like, he was able to derive his laws of motion using parameters (e.g., position and velocity) that originated from those defined concepts. In a complex system, however, what may first appear to be an advantage may actually be a disadvantage. To demonstrate this, assume we use an ideal, simplified model of "spherical chickens in a vacuum" to perform computation — this model cannot actually solve any real-world problems. Given that data acquisition, storage, and computing have all become exceptionally convenient today, is it now possible to further explore the value of information in a more effective and efficient way than if we were to use traditional physical modeling methods? We will explore this possibility by analyzing traditional applied mathematics — a subject that closely combines symbols and experiments. It is comprised of the following five main steps:

- Create a mathematical model for a domain problem.
- Find the analytical or numerical method for solving the problem.
- Convert the problem-solving process into a computer algorithm.

- Compile the algorithm as a software program.
- Optimize the algorithm and software iteratively, and implement large-scale parallelization.

2.1 Creating a Mathematical Model for a Domain Problem

Data is a product of "intelligence." But despite data storing the essence of intelligence information, it does so in quite a fragmented way. Due to the lack of effective methods for capturing salient information from data, human civilization has struggled throughout history. Traditional mathematical modeling primarily involves finding abstract equivalent representations of things, such as concept definitions, hypothetical conditions, and mathematical relations. Even though the study of mathematical relations attracts the attention of most scholars, concept definitions and hypotheses essentially determine the quality and significance of traditional mathematical modeling. Let's look at two typical examples — Kepler's Laws of Planetary Motion and Newton's Laws of Motion. Using a geometric language conceived by René Descartes, Kepler's Laws explain the motion of planetary bodies through the mathematical relations of several concepts such as planetary coordinates, time, and orbital parameters. Newton's Laws go one step further by coining the terms of mass, action relationship, and acceleration. Among them, the concept of "mass" unifies the measurement of all substances, meaning that all objects — from those as big as the Sun and the Moon to those as small as apples and oranges — can be measured by just one dimension, namely, their mass. Such symbolic abstractions, both mathematical and physical ones, are adopted by many scientific masterpieces, including Euclid's Elements written before antiquity, Newton's Mathematical Principles of Natural Philosophy in the 17th century, and Einstein's General Relativity in the 20th century. Similarly, Galois' group theory proposed in the 19th century unifies the algebraic structures with the concept of group. Later, quantum mechanics and quantum electrodynamics emerged to unify the action relationships between inorganic matters in nature through four fundamental interactions. This kind of representation, achieved in an "equivalent" or "unified" manner, has demonstrated remarkable advantages in the induction, deduction, and spreading of science throughout human civilization.

At the substance level, equivalent representation of action relationships has brought fundamental changes to society. Yet at the consciousness (or human intelligence) level, we still lack a unified representation that proves effective for modeling. On the one hand, it is impossible to acquire parameters from within the system — in this case, the human brain. On the other hand, rules summarized by humans do not perform well enough to address problems in open domains. For instance, in the early stages of development (in the 1950s), machine translation adopted a grammar rule–based approach (i.e., one based on expert knowledge) and consequently struggled to develop further due to the inefficiency of rule-based systems. It was not until data-driven statistical methods were employed in the 1990s that machine translation began to deliver satisfactory performance. Although such modeling methods drove breakthroughs in speech recognition and text recognition, their value was not fully realized at that time because the capabilities of storing, reading, and processing big data still needed to be perfected. Later in 2012, data-driven neural networks emerged and demonstrated compelling performance in classification. Since then, statistical modeling based on neural networks has been creating unprecedented value in many fields.

The rapid development of neural network methods leads us to ask ourselves: What makes AI technologies so special? And what are the advantages and disadvantages of AI methods with respect to their technical principles? The currently popular "Datasets + Neural network architecture + Gradient backpropagation" approach is actually an automated "Modeling + Training + Generalization & inference" framework designed for specific high-dimensional problems. The core philosophy of such a framework in terms of problem solving is "to make an assumption and verify it."

As an example, let's look at the modeling of an image classification problem. For a given set of images, the input dimensionality is $d = 3 * 224 * 224$, and an expectation function based on L1 loss is used as the target function. According to the universal approximation theorem, the approximation error of neural network fitting is independent of the input dimensionality. We can therefore use neural networks to model high-dimensional problems [1]. Weinan E et al. also analyzed in further detail the performance of two-layer neural networks and that of residual networks in Barron space and flow-induced function space separately [2].

Gradient disappearance and dispersion are two factors that we may encounter during the training of neural networks. However, we can address them by respectively selecting an appropriate activation function (e.g., the popular ReLU function) and normalizing parameters. For a neural network that has many layers, the training effect may not meet expectations — in this case, we can use a residual neural network (ResNet) to ensure the relevance of gradients [3]. In the case of an L-layer network, the attenuation of gradient relevance in a PlainNet without residual representation is $\frac{1}{2^L}$, and that of a ResNet is only $\frac{1}{\sqrt{L}}$ [4]. A ResNet is also more flexible in terms of its representation capability, and performs especially well in the selection and combined use of shallow and deep features. With these advantages, a ResNet is more suitable for learning from large datasets. From this, we can conclude that there is very little logical reasoning or deduction involved in the process of fixing training issues. Instead, theoretical analysis is often conducted on a solution that has been verified as valid through experiments. As such, we can determine that the ResNet representation of image classification problems provides us a better loss landscape [5] and considerably reduces the difficulty involved in gradient optimization. Even so, these advantages have yet to be theoretically proven in a strict sense.

Nevertheless, the main difficulty encountered in data-driven and AI-based statistical modeling lies in the generalization capability of models. Even if the previous training issues are addressed, the resulting models will still depend strongly on training data. Intuitively, we can see that the difficulty of generalizing a model goes up as the actual environment of inference and the training data become more and more dissimilar. The generalization predicament also occurs in traditional physical modeling — the most typical example of this is that Newtonian mechanics at the macro level does not apply to quantum mechanics at the micro level. Things become worse or even disastrous when we perform inference with an AI-based neural network model. This is because we cannot determine the scenario applicability (i.e., the generalization capability) of a model until the results of inference are verified. Theoretically, we can use the Rademacher complexity to describe a model's capability of fitting random noise on a given dataset. However, the bound provided by the Rademacher complexity can barely match real-world scenarios. Such difficulties are major factors that hinder the generalization of models. Is there a way to combine established science with big data–based AI technologies that are widely used today, so that we can find a "unified" information processing technology that brings higher efficiency and hence alleviate — to the largest degree practicable — the model generalization predicament of high-dimensional problems? Without a ready answer to this question, we can still study the following successful cases in industry to see whether they offer us some insights, either physically or mathematically.

- In what way is the Transformer architecture [6], which adopts a self-attention mechanism, similar to the action mechanism of universal gravitation and Coulomb force?
- How can we guarantee the equivariance of 3D roto-translations [7] during the training of a neural network?
- Using side length restrictions of triangles, can we improve the method of training agent models for protein structure generation [8], such that AlphaFold2 will reach a computational accuracy comparable to experimental results?
- Can momentum knowledge be leveraged to improve gradient optimization algorithms [9]?

Using implicit knowledge in big data to facilitate modeling remains an open and long-term undertaking. Now we explore several typical unsolved industrial problems, for which using the AI approach is expected to achieve breakthroughs.

OPC Modeling

To model the optical proximity correction (OPC) process of a lithography machine, for instance, the traditional method builds a physical model for each lens and then stacks these models. This method, while allowing us to obtain adequate results under relatively low accuracy requirements, is no longer effective when lenses approach their optical limits and the system becomes notably affected by factors such as temperature and lens imperfections. One possible approach of overcoming this issue is to construct a hybrid model that combines the deep neural network model and the traditional physical model. This approach has been preliminarily verified as effective — as such, it is fair to say that introducing AI methods in traditional modeling may bring very good results.

Optical Communication Module Modeling

In an optical-electrical conversion system used in optical communication, the presence of non-linear physical signals can cause dynamic damages and consequent coupling of the damages. As a result, the classical method of signal processing and modeling is unable to accurately model the signal conversion process and often produces a high bit error rate. Using AI technologies can help rectify the defects found in the physical models created for optical communication. With the novel AI approach, we can treat an optical-electrical conversion system as a non-linear waveform transformation function. By using it, we can directly predict waveform transformation and therefore parse specific resulting effects based on the predicted waveform. Furthermore, AI can detect non-linear signals in signal patterns during the early stages of simulation, making it possible to apply a compensation algorithm accordingly. AI detection is much more accurate and efficient than manual identification, especially when we consider the microscopic nature of non-linear signals and the difficulty in intuitively perceiving such signals.

Astronomical Phenomena Modeling

In astronomy, radio telescopes like China's FAST (Five-hundred-meter Aperture Spherical Telescope) have profound capabilities of receiving and collecting signals, enabling the discovery of many astronomical phenomena such as pulsars and radio storms. Despite this, the signal processing capability of these telescopes falls far behind the speed at which they gather information — a performance bottleneck. In traditional signal processing, a radio telescope builds a physical model in the form of a time-frequency spectrogram that involves an unknown distance parameter. Waves of different frequencies spread at varied speeds, and these waves present distinct slopes in spectrogram signals. Leveraging these characteristics, scientists are able to determine the astronomical distance through a grid traversal of dispersion values or downsampling values, or by performing Gaussian fitting on the time axis. Nevertheless, this approach involves very high computational costs and is extremely inefficient.

Today, AI technologies offer the prospect of solving this problem. By labeling deterministic astronomical signal samples and learning from the labeled samples, we can directly determine signal waveforms in a graphical manner, enabling us to solve the astronomical distance based on the waveforms. This kind of problems involve a large amount of unlabeled data, in which case semi-supervised AI learning has demonstrated good performance. For example, a "distillation learning" algorithm can learn and train on de-dispersed data, and then transfer the resulting model to learn on non-de-dispersed data in order to provide predictions. Because blind search becomes feasible without the need to consider dispersion, such an algorithm considerably speeds up searches and can even realize real-time search. Likewise, semi-supervised positive-unlabeled learning (or PU learning) algorithms can effectively explore a large amount of unlabeled data at a reasonably low training cost by mixing the unlabeled data with labeled samples. PU learning algorithms have yielded discoveries of valid pulse signals and achieve a higher detection efficiency.

2.2 Problem Solving

Even with accurate physical modeling, solving high-dimensional problems can still be catastrophic due to the absence of efficient methods. For instance, a method as powerful as the Schrödinger equation can only solve hydrogen atom problems accurately. To address this issue, academics have developed many approximation algorithms, for example, the density functional theory. The tradeoff between accuracy and computational complexity is always a thorny issue because we can never have both. As such, applying neural networks to solve high-dimensional partial differential equation (PDE) problems has naturally become a hot subject of research. This research is being conducted in the following two main directions.

First, we can use the dataset obtained through a traditional numerical solver to train a deep neural network, and then use this network as an agent simulator to achieve higher solving performance [10]. However, this approach does not allow theoretical analysis on accuracy bounds or convergence of solutions. Consequently, it does not provide any theoretical guarantee. In other words, this method provides only speculations for certain tasks that traditional methods fail to solve [11–13].

Second, we can use neural networks to learn mapping relationships between functions, and then use these mappings as operators for solving PDE problems. The two most typical research efforts in this regard are DeepONet [14] and the Fourier neural operator [15]. Once they are learned for an entire category of PDEs, these operators can be used directly for different initializations or bounds, without the need for retraining to be performed. Nonetheless, the specific PDE categories to which such operators are applicable still depends on further theoretical analysis [16, 17].

3 A Novel Method of Integrating Computer Graphics and Multimedia Technologies

In computer graphics, an image is constructed and rendered based on a 3D geometric model in a coordinate system. Pixels are generated to form a corresponding image within geometric surfaces and grids, from a certain point of view or under specific illumination conditions. In multimedia processing, however, an image is generated after information collected from the real world is processed by the color filter array and sensors, followed by a series of post-processing (such as de-mosaicization, white balance, and gamma correction). These two image generation methods have developed in parallel over the course of more than two decades. Only now though is it possible to integrate them thanks to the introduction and popularization of AI technologies.

Traditional computer graphics renders images on the basis of mathematical and physical modeling that utilizes the optical process (e.g., rasterization and ray tracing) of camera photographing. To approach this differently, the neural radiation field (NeRF) [18–20] and multiplane image (MPI) [21–23] techniques, along with many of their subsequent works, use deep learning for image rendering. In this case, the rendering process directly treats the model of 3D objects and their scenario as a function, which takes incident rays as input values and provides an output of reflected light rays with specific color, intensity, and direction information to represent an image. By learning functional mapping through AI, 3D models and images can be directly connected to each other without the use of geometric elements. This integrated approach creates new possibilities for further development of multimedia, graphics, and even metaverses.

4 A Groundbreaking Paradigm of Developing Autonomous Robotics

Given the enormous applications of AI, it is only natural for us to think how we can enable more intelligence for machines or robots. Today's robots possess only simple and limited intelligence — their level of autonomy is far too weak to even complete tasks independently. Traditionally, we control robots by modeling their states and the environment in which they work based on a precise coordinate system. To put it differently, by using a robot's position, action, and state information collected through sensors and sensing algorithms, we can train a model with high-dimensional coordinates. As such, we can then use the model to predict and plan the robot's actions and the exact way in which each action happens (i.e., the path, distance, angle of movement, and so on). The planned information (i.e., instructions for the robot's next action) is then passed to the PID (Proportional-Integral-Derivative) controller for execution, with collaboration from the motor controller. This inefficient control process drastically limits the capabilities of robots compared to humans and animals.

Now that AI has demonstrated its breakthroughs in computer graphics, we can also apply AI methods in robot control. In other words, we can train an end-to-end, graph-based control model that does not require the use of a coordinate system — similar to the case with computer graphics we mentioned earlier. Using this model, a robot can directly take actions based on what it has observed and learned, much the same as a living creature does. For example, if we want a robotic arm to strike a ball like a professional baseball player, we can train the arm by repeatedly throwing baseballs at it — with a certain degree of variation in how it is thrown — so that the arm learns how to hit each baseball perfectly. A robotic arm trained in this way can strike a baseball on par with a professional in real baseball games. Most importantly, such training has been preliminarily proved as being effective.

5 A Fresh Understanding on the Learning and Reasoning of Structured Knowledge

In the Internet age, big data usually comes in the form of dynamic, unstructured, but strongly correlated texts. Many experts renowned in the field of AI do not advocate adopting the structured approach of knowledge learning. Instead, they believe that demand-driven, fragmented information ingestion benefits AI learning better and can yield an optimal learning effect. That being said, the human race over several thousands of years of civilization has adopted a process of extracting and accumulating structured knowledge from fragmented information. In particular, the emergence of languages and characters — as a typical and well-known structured symbol system — has profoundly accelerated the transmission of information, virtually broken the limitations of time and space in knowledge transfer, and substantially pushed our ability to exponentially accumulate knowledge. Therefore, learning and reasoning of structured knowledge could be the ultimate problem of AI development.

The development of AI relies on data, algorithms, compute power, and applications. Among these four factors, data is usually sufficient, compute power is attainable, and AI application scenarios are broad enough to guarantee a promising future, whereas AI algorithms now turn out to be a major bottleneck. Learning and reasoning of structured knowledge can be leveraged to effectively address this bottleneck, helping us to achieve AI algorithm breakthroughs.

A knowledge learning system can use a weighted mapping network to efficiently express structured knowledge, which maps to specific objects in real life. Such a system should possess some kind of "memory" and use this memory to continuously store new knowledge. Using the stored knowledge, the system should be able to perform repeated reasoning on new problems and eventually solve them.

In the industry, many ultra-large models were proposed on the basis of the Transformer architecture mentioned in Section 2.1. As an example, the GPT3 model [24] launched in 2020 adopted a new prompting approach and demonstrated human-level capabilities in many humanities areas, such as writing articles and even poetry. Yet even a model as large as GPT3 could not adequately understand certain structured knowledge, of which mathematics is a very typical subject. Performance evaluations of [25] showed that models had only near-random accuracy for most science, technology, engineering, and mathematics (STEM) tasks.

Just one year later, however, researches utilizing human-like thinking (e.g., chain-of-thought prompting) emerged [26, 27], followed by works such as the Pathways Language Model (PaLM) [28] — a large language model that achieved scaling through the use of Pathways. Based on these works, the latest Minerva model [29] improved the accuracy of the MATH dataset (mathematics competition level for high school) [30] from 6.9% to 50.3% — compared to only 40% of an average high school student, after learning more than 1.2 million arXiv papers in LaTeX format and a large amount of mathematical knowledge in webpages.

Mathematical symbol–based calculation and theorem proving have been effectively contributing to improving AI's reasoning and prediction abilities. To learn the calculation of mathematical symbols and theorem proving, AI can adopt a human-like learning approach in order to solve problems with limited knowledge and complex reasoning skills. This ability, which extends limited knowledge to infinite applications, will enable AI to achieve automated exploration, learning, and prediction in many more open fields. Currently, AI has excelled in some simple calculation and theorem proving, such as symbolic calculation of trigonometric functions [31] and solving elementary school–level mathematical word problems. We expect that, in the future, AI will prove all theorems of advanced mathematics in a fully automated way. While this accomplishment will make AI capable of applying the knowledge it has learned, other optimization methods will take it to an even higher level. With theorem representation optimization through a hypertree [32], or with learning method optimization based on the GPT-f model [33] and Expert Iteration [34], Mathlib's tested inference accuracy for pass@1 has reached 63%, and a small number of International Mathematical Olympiad (IMO) contest questions have been successfully proved [34].

This AI approach based on structured knowledge also exhibits potential in solving open-domain problems. For example, an AI-based search engine can directly provide answers to some questions that require reasoning. This kind of search engine can actually be used as a chatbot [35], although such application may come at the expense of certain quality defects or sensitive issues. Such an AI approach kicks off the first yet critical step in solving open-domain problems.

6 Material Basis of AI

The development of AI naturally relies on a material foundation. Throughout the development of traditional CPUs, the material basis has involved a complete system of single-chip microcomputers, PCs, servers, and supercomputing centers. Likewise, humanity as a whole needs to work together and establish a full AI material system from 10-cent single-chip microcomputers to 1-billion-dollar supercomputing centers, in order to firmly sustain the rapid development of AI.

References

[1] Y. Lu and J. Lu, "A universal approximation theorem of deep neural networks for expressing distributions," *CoRR*, vol. abs/2004.08867, 2020. [Online]. Available: https://arxiv.org/abs/2004.08867

[2] W. E, C. Ma, and L. Wu, "Barron spaces and the compositional function spaces for neural network models," *CoRR*, vol. abs/1906.08039, 2019. [Online]. Available: http://arxiv.org/abs/1906.08039

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[4] D. Balduzzi, M. Frean, L. Leary, J. P. Lewis, K. W. Ma, and B. McWilliams, "The shattered gradients problem: If ResNets are the answer, then what is the question?" *CoRR*, vol. abs/1702.08591, 2017. [Online]. Available: http://arxiv.org/abs/1702.08591

[5] H. Li, Z. Xu, G. Taylor, and T. Goldstein, "Visualizing the loss landscape of neural nets," *CoRR*, vol. abs/1712.09913, 2017. [Online]. Available: http://arxiv.org/abs/1712.09913

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," CoRR, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[7] F. Fuchs, D. E. Worrall, V. Fischer, and M. Welling, "SE(3)-Transformers: 3D roto-translation equivariant attention networks," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/15231a7ce4ba789d13b722cc5c955834-Abstract.html

[8] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, and D. Hassabis, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, pp. 1–11, 08 2021.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[10] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part I): Data-driven solutions of nonlinear partial differential equations," *CoRR*, vol. abs/1711.10561, 2017. [Online]. Available: http://arxiv.org/abs/1711.10561

[11] J. Han, A. Jentzen, and W. E, "Overcoming the curse of dimensionality: Solving high-dimensional partial differential equations using deep learning," *CoRR*, vol. abs/1707.02568, 2017. [Online]. Available: http://arxiv.org/abs/1707.02568

[12] H. Wang, L. Zhang, J. Han, and W. E, "DeePMD-kit: A deep learning package for many-body potential energy representation and molecular dynamics," *CoRR*, vol. abs/1712.03641, 2017. [Online]. Available: http://arxiv.org/abs/1712.03641

[13] L. Fang, P. Ge, L. Zhang, H. Lei, and W. E, "DeepN^2: A deep learning-based non-newtonian hydrodynamic model," *CoRR*, vol. abs/2112.14798, 2021. [Online]. Available: https://arxiv.org/abs/2112.14798

[14] L. Lu, P. Jin, and G. E. Karniadakis, "DeepONet: Learning nonlinear operators for identifying differential equations based on the universal approximation theorem of operators," *CoRR*, vol. abs/1910.03193, 2019. [Online]. Available: http://arxiv.org/abs/1910.03193

[15] Z. Li, N. B. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. M. Stuart, and A. Anandkumar, "Fourier neural operator for parametric partial differential equations," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=c8P9NQVtmnO

[16] S. Lanthaler, S. Mishra, and G. E. Karniadakis, "Error estimates for DeepONets: A deep learning framework in infinite dimensions," *CoRR*, vol. abs/2102.09618, 2021. [Online]. Available: https://arxiv.org/abs/2102.09618

[17] N. Kovachki, S. Lanthaler, and S. Mishra, "On universal approximation and error bounds for Fourier neural operators," *J. Mach. Learn. Res.*, vol. 22, pp. 290:1–290:76, 2021. [Online]. Available: http://jmlr.org/papers/v22/21-0806.html

[18] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Commun. ACM*, vol. 65, no. 1, pp. 99–106, 2022. [Online]. Available: https://doi.org/10.1145/3503250

[19] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *CoRR*, vol. abs/2201.05989, 2022. [Online]. Available: https://arxiv.org/abs/2201.05989

[20] M. Tancik, V. Casser, X. Yan, S. Pradhan, B. Mildenhall, P. P. Srinivasan, J. T. Barron, and H. Kretzschmar, "Block-nerf: Scalable large scene neural view synthesis," *CoRR*, vol. abs/2202.05263, 2022. [Online]. Available: https://arxiv.org/abs/2202.05263

[21] T. Zhou, R. Tucker, J. Flynn, G. Fyffe, and N. Snavely, "Stereo magnification: Learning view synthesis using multiplane images," *ACM Trans. Graph.*, vol. 37, no. 4, p. 65, 2018. [Online]. Available: https://doi.org/10.1145/3197517.3201323

[22] B. Mildenhall, P. P. Srinivasan, R. O. Cayon, N. K. Kalantari, R. Ramamoorthi, R. Ng, and A. Kar, "Local light field fusion: Practical view synthesis with prescriptive sampling guidelines," *ACM Trans. Graph.*, vol. 38, no. 4, pp. 29:1–29:14, 2019. [Online]. Available: https://doi.org/10.1145/3306346.3322980

[23] J. Flynn, M. Broxton, P. E. Debevec, M. DuVall, G. Fyffe, R. S. Overbeck, N. Snavely, and R. Tucker, "DeepView: View synthesis with learned gradient descent," in *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 2367–2376. [Online]. Available: http://openaccess.thecvf.com/content_CVPR_2019/html/Flynn_DeepView_View_Synthesis_With_Learned_Gradient_Descent_CVPR_2019_paper.html

[24] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[25] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=d7KBjmI3GmQ

[26] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. H. Chi, Q. Le, and D. Zhou, "Chain of thought prompting elicits reasoning in large language models," *CoRR*, vol. abs/2201.11903, 2022. [Online]. Available: https://arxiv.org/abs/2201.11903

[27] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," *CoRR*, vol. abs/2203.11171, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2203.11171

[28] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V.

Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *CoRR*, vol. abs/2204.02311, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2204.02311

[29] A. Lewkowycz, A. Andreassen, D. Dohan, E. Dyer, H. Michalewski, V. Ramasesh, A. Slone, C. Anil, I. Schlag, T. Gutman-Solo, Y. Wu, B. Neyshabur, G. Gur-Ari, and V. Misra, "Solving quantitative reasoning problems with language models," 2022.

[30] D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt, "Measuring mathematical problem solving with the MATH dataset," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html

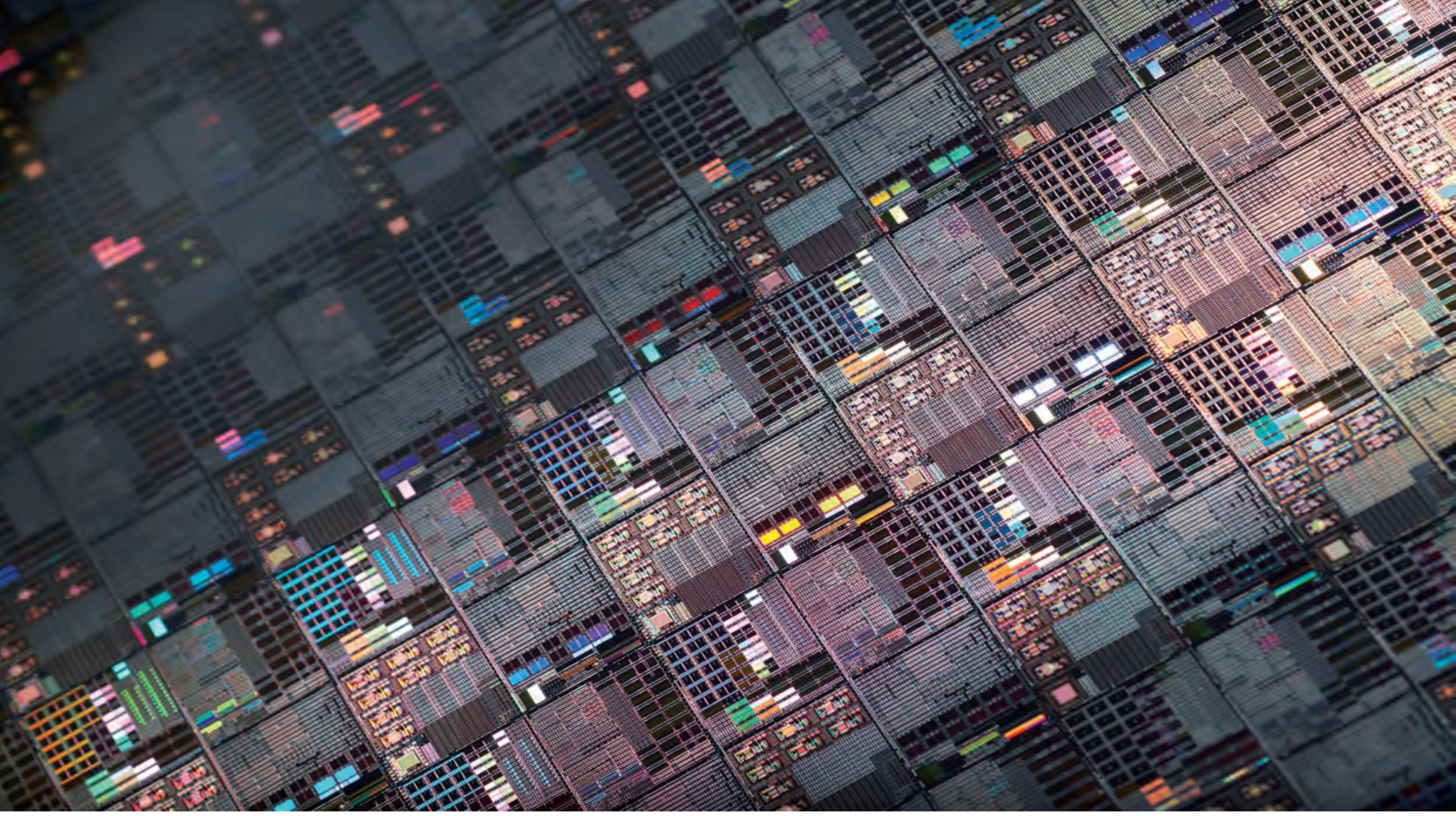
[31] Z. Liu, Y. Li, Z. Liu, L. Li, and Z. Li, "Learning to prove trigonometric identities," *arXiv e-prints*, p. arXiv:2207.06679, Jul. 2022.

[32] G. Lample, M. Lachaux, T. Lavril, X. Martinet, A. Hayat, G. Ebner, A. Rodriguez, and T. Lacroix, "Hypertree proof search for neural theorem proving," *CoRR*, vol. abs/2205.11491, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2205.11491

[33] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," *CoRR*, vol. abs/2009.03393, 2020. [Online]. Available: https://arxiv.org/abs/2009.03393

[34] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever, "Formal mathematics statement curriculum learning," *CoRR*, vol. abs/2202.01344, 2022. [Online]. Available: https://arxiv.org/abs/2202.01344

[35] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, Y. Zhou, C. Chang, I. Krivokon, W. Rusch, M. Pickett, K. S. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. H. Chi, and Q. Le, "Lamda: Language models for dialog applications," *CoRR*, vol. abs/2201.08239, 2022. [Online]. Available: https://arxiv.org/abs/2201.08239



Spatial Computing: Scalable Computing Systems Based on Computational Graphs

Tianqi Wang

Abstract

Semiconductor manufacturing is still developing but results are being delivered at a slower pace. Emerging applications, like AI, are demanding exponentially more and more computing power. Processors based on the traditional von Neumann architecture are suffering from the constraints of the memory wall and power wall. And the overhead involved in maintaining the shared-memory model is increasing rapidly, making it more difficult to scale out computing systems. These factors make it impossible to meet the demand of applications for computing power. In terms of scalability, spatial computing — which uses the distributed memory system — is increasingly used to build high-performance computing systems. This paper summarizes the existing progress and challenges of spatial computing and briefly introduces HiSilicon's ongoing research on the applications, programming models, and computational graph partitioning and mapping of spatial computing.

Keywords

spatial computing, high performance computing, AI

1 Introduction

Research into computer architecture has continued to advance as semiconductor manufacturing processes and application algorithms evolve. And the advancement of semiconductor manufacturing processes following Moore's Law is facilitating transistor manufacturing. Moreover, growing application algorithms manage to convert the computing power of transistors into actual productivity. On top of that, research into computer architecture has led to the development of more efficient organization methods, paving the way for transistors to achieve more effective computing power.

Spatial computing is one of the most valuable research directions of computer architectures. This paper illustrates the necessity of application of "More than Moore" — a new paradigm for computing power growth — in the research of computing system architecture from the application algorithm and manufacturing process perspectives. It also dives into the potential of spatial computing, an architecture of "More than von Neumann" that might provide extra computing power in the future. The paper then summarizes the current work and challenges in spatial computing architecture, and outlines the spatial computing research of the HiSilicon research team in fields such as application scenario analysis, programming model, and computational graph deployment.

1.1 More Than Moore

Moore's Law observes a virtuous cycle of economy where user consumption drives investment, which then fuels technological progress, and then the resulting product improvements and upgrades attract further user consumption. However, the development of semiconductor manufacturing processes lacks fuel. As such, the virtuous cycle is on the brink of collapse. That is, the slowdown of technological progress attracts fewer new users, making it difficult to draw enough investment in technological progress, and this in turn results in slower technological progress. Figure 1 illustrates these virtuous and vicious cycles.

As mentioned earlier, Moore's Law is essentially a commitment to stable growth that attracts continuous investment from social resources. What unequivocally attracts continuous social investment is not the growth of nominal parameters, but that of available computing power — essentially, the better life it brings. Thus, computing power is facing mounting challenges in cutting-edge fields related to the national economy and people's livelihoods. This is evident in numerous cases. For example, materials and drug design relies on the computational processing of molecular dynamics or computational chemistry to simulate system evolution [2]. The design and control of internal combustion engines depend on accurate hydrodynamics and structural mechanics modeling [3]. Accurate weather forecast and earthquake warning cannot be divorced from fast and precise geophysical simulation [4]. And accurate simulation and real-time control of electromagnetic fields make controlled nuclear fusion possible [5].

Numerical solutions of partial differential equations, such as the finite element method, have made outstanding achievements in scientific computing. But today, these conventional methods are not advanced enough to handle sophisticated issues, such as simulating complex chemical reactions or phenomena of turbulence and shock waves, and modeling phase transition processes. They fail to generate accurate computing results for large-scale systems and even get stuck during the initial stages. The major obstacle lies in the complex system and physical process as they depend on too many variables. When high-dimensional functions are used to represent the system, the curse of dimensionality arises, resulting in the inability to solve problems. As shown in Figure 2, the performance of the top 500 supercomputers has improved 1000-fold in the space of 10 years, from 2005 to 2015. If three-dimensional grids had been used to solve partial differential equations, only a tenfold increase of the grid resolution could have been achieved over the same 10-year period, not to mention the complex algorithm of the equation solver, which could further lower the final resolution improvement. *The curse of dimensionality is substantially rooted in function approximation, a*

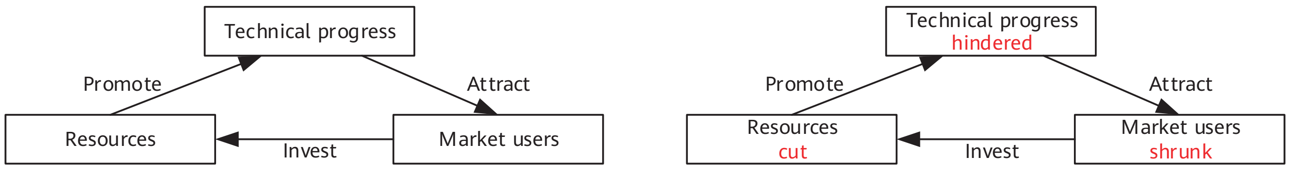


Figure 1 Virtuous cycle and vicious cycle of Moore's Law [1]

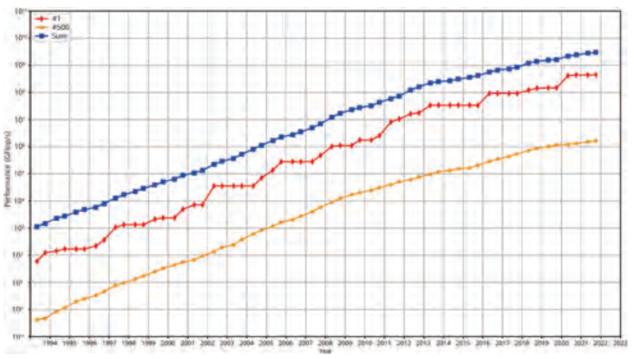


Figure 2 Computing power survey of the top 500 supercomputers [6]

conventional method to handle issues with high complexity and low generalization based on polynomial functions and piecewise functions.

The AI method has been successfully applied in extensive fields such as image classification and natural language processing. At the same time, the AI method has been widely introduced into the scientific computing fields mentioned earlier and achieved a series of remarkable achievements. From the point of view of computational mathematics, the AI method provides an effective tool to solve high-dimensional mathematical problems. Given that the target approximation error is ϵ, the dimension of the problem to be solved is d, and the total number of parameters required for approximate modeling is m. The convergence rate of the traditional polynomial approximation will be $\epsilon \sim m^{-1/d}$. That is, when the error $\epsilon \sim 0.1$, the total number of parameters required is $m \sim 10^d$. However, the convergence rate of AI modeling based on neural networks is $\epsilon \sim m^{-1/2}$. That is, when the error $\epsilon \sim 0.1$, the total number of parameters required is only $m \sim 10^2$.

The application scope of the AI method is limited due to its intrinsic nature of inexplicability. However, the AI method can contribute to solving some highly complex problems that other methods cannot solve. For example, analytical methods cannot simulate the scale of meaningful problems (such as first-principle simulation of organic chemical reactions) [2] within an acceptable period of time. Another example is that current analytical methods provide unstable solutions, requiring the introduction of many phenomenological methods (such as simulation of turbulent flows and phenomenological theory in high energy physics) [3]. These phenomenological methods depend on the experience or intuition of scientists, while the contemporary AI method can systematically improve the expression capability of empirical phenomenological models. In addition, the AI method is widely used in engineering practices that need to control the experimental system in real time. For example, in a magnetic confinement fusion experiment that mandates accurate control of the magnetic field position, the conventional high-precision electromagnetic field solution cannot achieve real-time control. Consequently, phenomenological models of many empirical parameters are used [5].

No pain, no gain, as the saying goes. Although the AI method has the potential to solve cutting-edge problems in science and engineering, it requires that larger-scale AI networks be built for complex systems such as natural language processing (NLP) and multiphysics. As shown in Figure 3, AI model sizes have increased by nearly four orders since 2018, posing new challenges to inference and training systems of AI networks. *This drives computing system designers to adopt "More than Moore" — a computing power growth paradigm much faster than Moore's Law based on existing semiconductor manufacturing processes.*

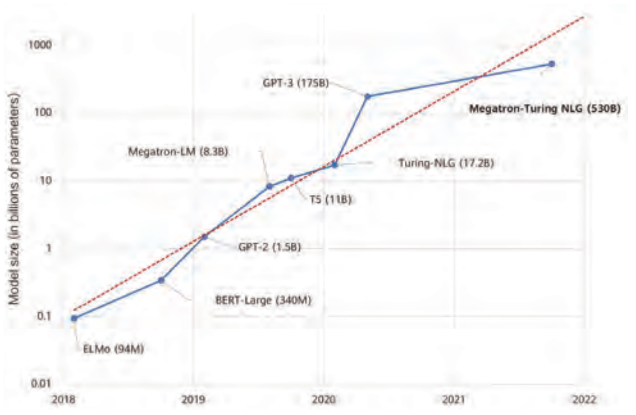


Figure 3 AI model size growth by time [7]

1.2 More Than von Neumann

Turing's paper in 1936 gave a precise mathematical description of the Turing machine and explored the boundary of the computable problem of the Turing machine [8]. As shown in Figure 4, the Turing machine consists of seven basic elements: a group of architectural states Q, a

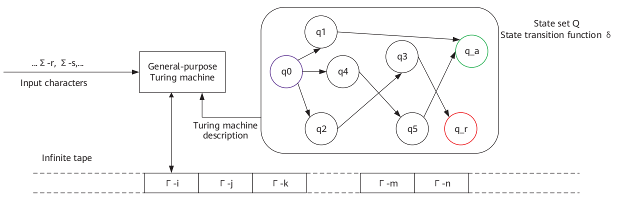


Figure 4 Basic elements of a general-purpose Turing machine

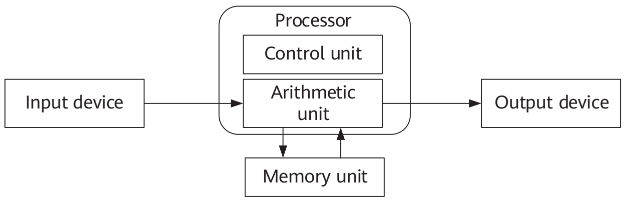


Figure 5 Traditional von Neumann architecture

set of input characters Σ, a set of paper tape characters Γ, a set of state transition functions δ, an initial state $q0$ within Q, an acceptance state q_{accept}, and a rejection state q_{reject}.

However, Turing did not dive into the implementation strategy of the Turing machine. It was von Neumann who proposed a realizable architecture of the Turing machine. In Figure 5, the von Neumann architecture defines a basic architecture including a control unit, arithmetic unit, and memory unit, meaning that it defines a method for implementing the aforementioned seven basic elements such as the architecture state Q and the state transition function δ in a Turing machine. This method has been highly applicable despite the constraints of the digital circuit technology and semiconductor technology at that time, and has become the cornerstone of traditional computer architecture research.

The success of Moore's Law stems from its commitment to stable growth. Accordingly, architecture designers must continuously improve the computing power of computing systems with a certain methodology. Based on the hardware processor of the von Neumann architecture, software developers have developed large and complex high-level software ecosystems, which are even more expensive than hardware systems. The traditional von Neumann architecture agrees on a set of interfaces for software and hardware, including the instruction set architecture (ISA) and shared memory model, and separates their design. As such, hardware and software can be developed independently and smoothly inherit the existing software. Within this system, the compiler acts as a software-to-hardware mapping tool to design the controllable complexity, facilitating the success of the traditional architecture.

As modern semiconductor manufacturing processes and packaging technologies advance, obvious limitations are becoming apparent in the von Neumann architecture. Discrete memory and computing units lead to the memory and power consumption walls. Centralized control units also restrict system scalability. Assume that an application requires N transistors, the available transistor density is N, and the footprint of the computing system is S. Then,

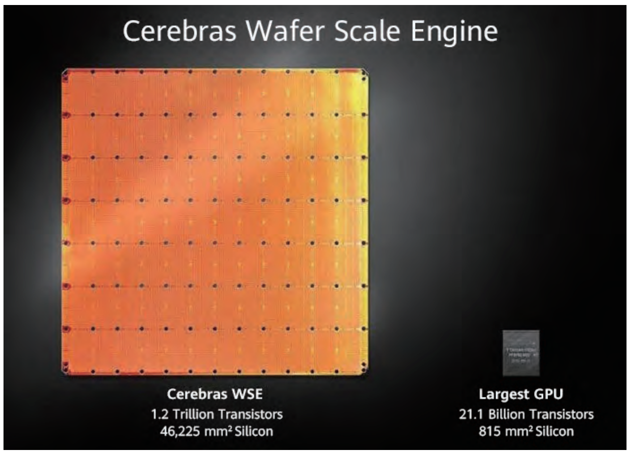
$$N = \rho \times S \qquad (1)$$

The growth rate of transistor scale dN is calculated based on those of transistor density $d\rho$ and computing system footprint dS

$$dN = d\rho \times S + \rho \times dS \qquad (2)$$

As mentioned earlier, the growth rate of unit transistor density resulting from Moore's Law is far from meeting the requirements of new applications such as AI ($dN \gg d\rho$ x S). This requires computer architecture researchers to effectively and reliably organize transistors (in pursuit of a larger dS) *in a larger footprint*.

The rapid development of packaging technologies is an attempt to organize transistors in a larger footprint. In recent years, chiplet and wafer-scale technologies have been widely used by major chip enterprises in their own products. Traditional processor providers such as AMD, Intel, and NVIDIA have produced CPU and GPU chips based on



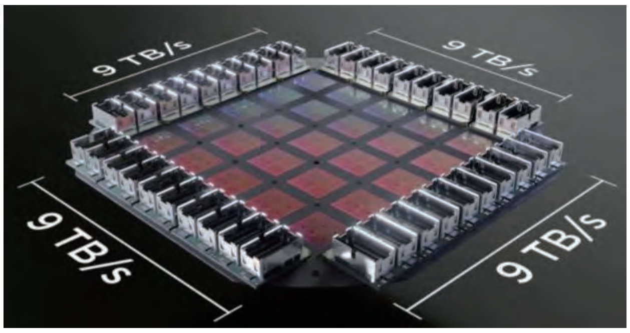


Figure 6 Cerebras CS-1 [9] (left) and Tesla DoJo [10] (right) based on the wafer-scale technology

the chiplet technology. Some startups and cross-border enterprises have also developed exploratory products based on the wafer-scale technology, such as Cerebras CS-1/2 and Tesla DoJo in Figure 6.

The shared memory model of the traditional architecture can inherit the existing high-level software. A method of implementing shared storage is to maintain a logically shared memory model on physically distributed storage data by using a consistency protocol (e.g., the CHI protocol [11] of ARM and the TSO [12] of Intel). The synchronization of status information initiated by the consistency protocol should be performed without requiring programmer involvement, meaning that the synchronization must be completed *per unit time*. In a traditional architecture, the unit time is usually at the level of nanoseconds, which is the execution duration of instructions.

High-density transistors following the Moore's Law prediction can continuously improve the on-chip interconnect bandwidth. However, the on-chip interconnect latency cannot be continuously reduced due to the constraints of physical laws. *In order to enlarge the chip size while trying to maintain the coherence and consistency to inherit the existing software ecosystem, the traditional von Neumann architecture framework must be abandoned. That means it is necessary to ease the existing coherence protocol to some extent, and to increase the granularity from the instruction level to the task level per unit time.*

Although this change can open up a new path for continuous growth of computing power, it also implies a cost on the software stack. That is, programmers need to develop new applications based on a task-level programming model, or refactor existing code to adapt to new computing power. In addition, software architects are also facing new challenges in how to design and develop a set of tools based on new hardware abstraction or configurations to effectively schedule and manage computing tasks. High performance computing (HPC) has been challenging the potential of computing scalability, and some inspiring attempts have emerged in this field. Despite the difficulty involved in automatically converting existing codes into task-based descriptions, OmpSs/OpenMP has designed some task-based compiler directive statements to migrate codes successfully [13]. In areas where algorithms such as first principles and computational chemistry evolve rapidly for more computing power, scientists are willing to pay the cost of refactoring code. NAMD/NWChem is a typical task-based programming model of code refactoring [13].

2 From Spatial Computing to Computational Graphs

2.1 Spatial Computing in a Broader Sense

Figure 7a shows the processor architecture spectrum. Processors based on the conventional von Neumann architecture usually *use centralized control logic and shared memory or register files, exchange data between arithmetic logic units (ALUs) by accessing the shared memory*, and perform tasks in a time sequence of instruction fetching, data fetching, and execution, as shown in Figure 7b. In addition to the time dimension, the spatial computing architecture also considers the spatial dimension. That is, many processing engines (PEs) are integrated into a given space in order to execute tasks at the same time. They have *independent control logic and distributed memories, and data directly flows between these PEs*, as shown in Figure 7c.

Traditional spatial computing architectures use many simple PEs to form arrays in order to achieve a high degree of parallelism of application algorithms. These PEs can directly communicate with each other in a point-to-point manner without detouring from a shared memory. Each PE may be independently programmed, and dataflows are formed between PEs. CGRA and FPGA are typical

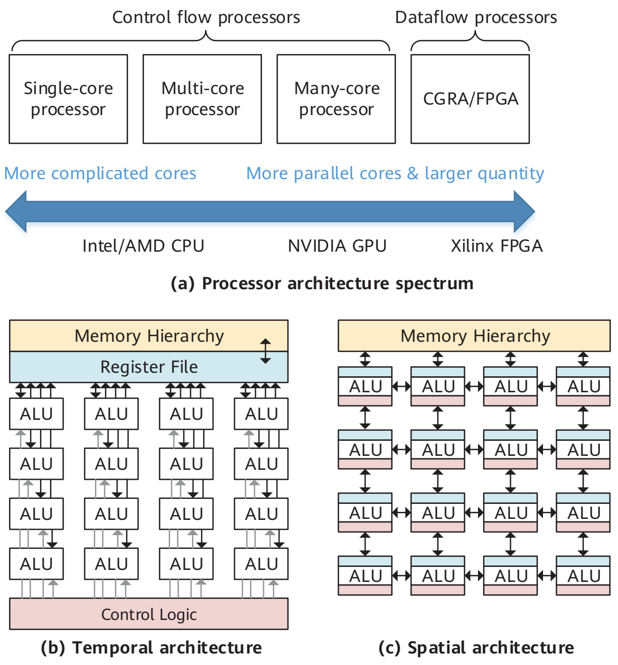


Figure 7 Processor system architecture spectrum [14]

examples of spatial computing architecture. Compared with the SIMD/SIMT architecture, the current spatial computing architecture processor is applicable to scenarios with significant producer-consumer dataflows (e.g., audio and video encoding and decoding, encryption, and communications).

As mentioned earlier, the decentralized control logic and distributed memory design of this architecture can fit into the physical implementation of hardware and achieve better scalability than the traditional von Neumann architecture does. However, in scenarios such as CGRA and FPGA, the PE granularity of a spatial computing processor is too small and lacks general programmability. It is difficult to carry a complex application with an obvious dynamic feature. Therefore, we attempt to expand the definition of spatial computing by replacing PEs with general-purpose processor cores in order to improve computing power and programmability. Our proposal ensures that multi-core processor clusters retain their dataflow design and work in pipeline mode. Spatial computing architectures such as CGRA and FPGA attempt to leverage fine-grained task-level parallelism in order to accelerate the computing process; the spatial computing architecture discussed in this paper aims to achieve better scalability using a similar distributed, decentralized design. This paper defines space computing in a broader sense — *it is dedicated to extending computing tasks to a larger space at the architecture level*. Therefore, this paper *focuses on scenarios that require high scalability, such as high performance computing and AI training*.

2.2 Abstract Layers of Computational Graphs

As shown on the left of Figure 8, research on the architecture is looking for a method M that maps a mathematical algorithm A to a physical implementation P, that is, $P=M(A)$. The traditional architecture, however, adds an abstraction layer (I) of ISA to divide the mapping into two phases — the software compilation phase (SW) from algorithm to ISA, and the hardware implementation phase (HW) from ISA to physical implementation, that is, $I = SW(M)$, and $P = HW(I)$.

Based on the previous discussion, research into computer architecture is facing the following major challenges:

- Although the AI method provides new tools that can solve real-world problems, it significantly increases the complexity of algorithms.
- The growth rate of transistor density slows down and new packaging technologies such as chiplet and wafer-scale technologies are introduced.
- As the complexity of algorithm A and physical implementation P increases dramatically, the mapping M from the algorithm to physical implementation is facing a dimensional explosion in complexity.

Complexity is an inexhaustible objective presence caused only by defects in architecture design. As an aphorism of David Wheeler goes, "All problems in computer science can be solved by another level of indirection" [15]. As shown on the right of Figure 8, there is an attempt to add an abstraction layer of the computational graph G between the ISA and the algorithm layer. That is, the algorithm M is mapped to the computational graph G through the graph compiler GC. The computational graph is converted into the ISA by the operator compiler OC. The hardware architecture completes the physical implementation, that is, $G = GC(M)$, $I = OC(G)$, and $P = HW(I)$.

In the new abstraction layer of computational graphs, the complexity of operators is shielded, and the dependencies between tasks as well as the input and output data are explicitly represented. Such a design coarsens the granularity of computation expressions of loads from the instruction level to the task level. In the traditional system architecture, the software compilation (SW) with dimensional explosion in complexity is divided into graph compilation (GC)

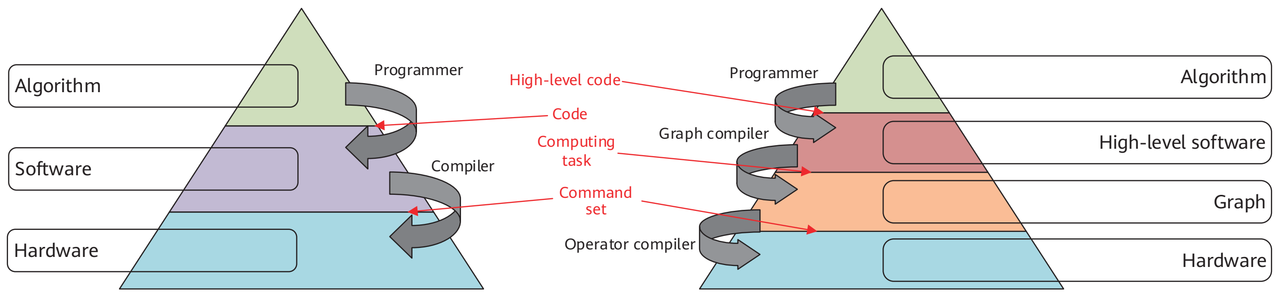


Figure 8 Abstract layers of the traditional system architecture (left) and spatial computing system (right)

and operator compilation (*OC*). In the phase of graph compilation, higher load granularity and explicit information can help graph optimization tools achieve efficient computing load deployment over a larger space.

3 Research Progress in the Industry

The industry has been exploring the boundaries of scalability with HPC for decades, providing significant reference value. The traditional cluster-level parallelism solution based on the programming model of message passing interface (MPI) and open multi-processing (OpenMP) has the following limitations in the HPC field:

- Poor programmability: MPI requires programmers to manually process data segmentation and load balancing, meaning that programming is reliant on the experience of a minority of professional programmers.
- Poor performance portability: Different supercomputing clusters have different hardware architectures. High scalability requires programmers to adapt an optimization solution to hardware.
- Difficult load balancing: Load imbalance, which causes low computing power utilization, occurs in strongly dynamic applications (such as sparse matrix computation, finite element analysis, and molecular dynamics simulation) when the process-thread model (i.e., MPI-OpenMP model) can only express coarse parallel granularity and the scheduling policy is fixed.

Due to these limitations, a series of graph-based solutions are proposed in the HPC field to bridge the gap between programmability and performance, and achieve better load balancing. The methods used in this field also involve programming models, runtime systems, and hardware acceleration. This section discusses several successful practices.

3.1 Legate

Legate, jointly developed by NVIDIA and Stanford University, can automatically deploy NumPy programs to heterogeneous distributed computing clusters. Legate is implemented on top of the Legion runtime system. Figure 9 [16] shows the Legate's rationale.

Programmers simply call NumPy to implement algorithms by means of serial programming. The compiler uses each NumPy function call as an operator to analyze data dependencies between NumPy function calls, and to generate a computational graph in order to show the dependencies. First, the computational graph is submitted to the Legion runtime system. Then, Legate Mapper maps the graph and makes it run on heterogeneous computing clusters of different scales. Experiment results show that although Legate significantly underperforms compared with the manually tuned MPI, programmers can leverage Legate

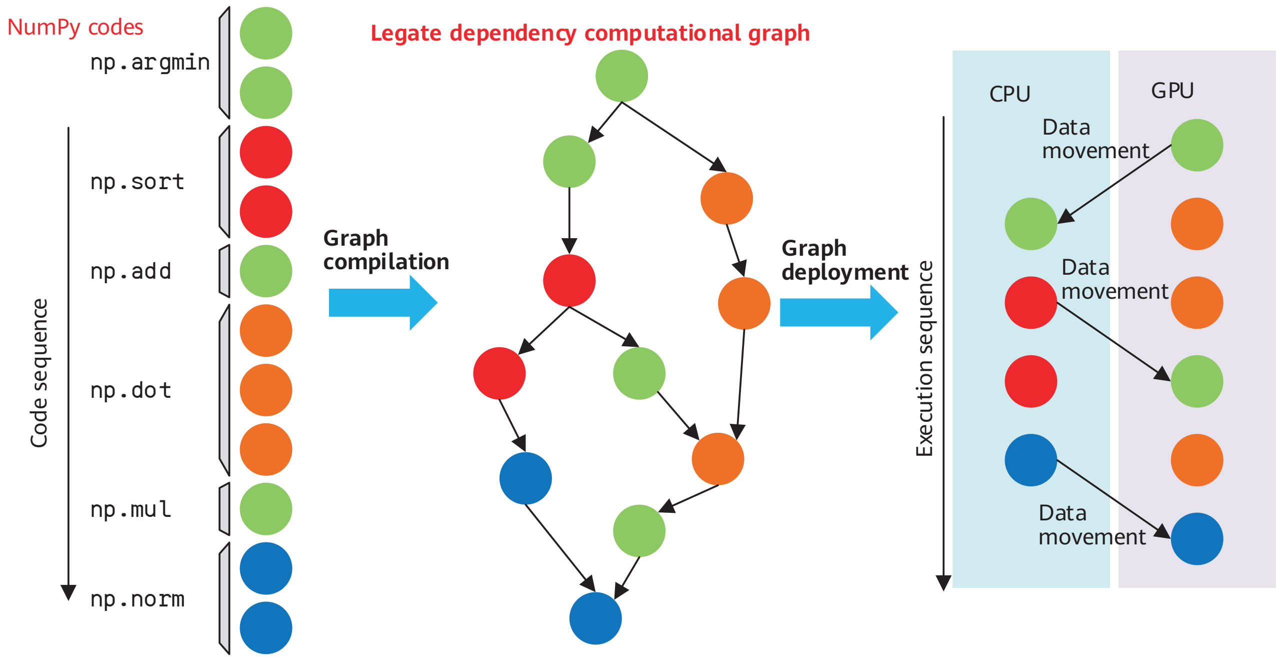


Figure 9 NumPy code sequence input by Legate (left), intermediate representation of Legate graph (middle), and physical deployment of Legate graph (right) [16]

to scale up from a single GPU to a cluster with multiple GPU enclosures, without needing to deal with complex parallel programming.

3.2 Charm++

Charm++ is a parallel programming framework developed by the University of Illinois Urbana-Champaign (UIUC). This object-oriented framework adopts the asynchronous messaging mechanism, as shown in Figure 10. Programmers can create a series of chare-objects that allow concurrent and asynchronous execution in Charm++. These objects can synchronize information by sending and receiving messages.

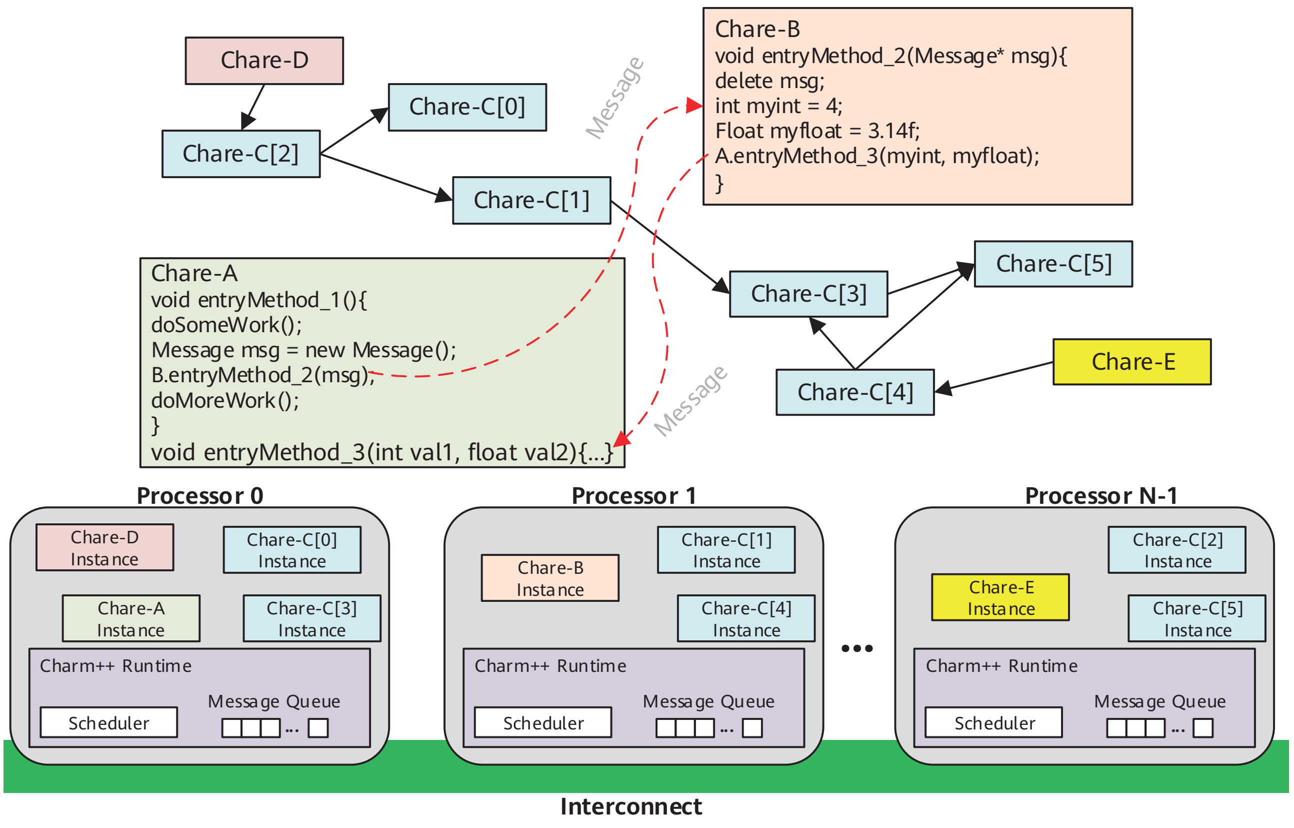
Implementations based on high-level semantics are compiled into computational graphs and submitted to the Charm++ runtime system, which deploys concurrent chare-objects (tasks) on multiple processors. Based on task status, the system automatically migrates each chare-object (task) among processors to achieve dynamic load balancing. A series of applications developed based on Charm++ have been widely used, especially NAMD, which is a molecular dynamics simulation software. Molecular dynamics simulations generally face dynamic load imbalances, resulting in poor scalability. However, the Charm++ framework helps improve the scalability of NAMD, outperforming other competitors that are also developed based on the MPI and OpenMP architecture [13].

In addition, Charm++ frees programmers of distributed hardware awareness and has the runtime/toolchain capable of solving the parallelization problem. A high-linearity program (such as LeanMD written in Charm++) can be implemented on a cluster containing more than 10,000 cores with only a few hundred lines of code. It largely facilitates verification of new algorithms for scientists.

3.3 Anton

Anton is a tailored computing chip developed by D. E. Shaw Research for molecular dynamics simulations. The computing cluster built on this chip achieves performance far beyond Summit, a supercomputing cluster. Figure 11 [18] provides details about Anton.

For different kernels involved in molecular dynamics simulations, Anton integrates heterogeneous computing units, specifically, customized accelerator pairwise point



```
Chare-B
void entryMethod_2(Message* msg){
delete msg;
int myint = 4;
Float myfloat = 3.14f;
A.entryMethod_3(myint, myfloat);
}
```

```
Chare-A
void entryMethod_1(){
doSomeWork();
Message msg = new Message();
B.entryMethod_2(msg);
doMoreWork();
}
void entryMethod_3(int val1, float val2){...}
```

Figure 10 Charm++ programming model based on concurrent chare-objects (above) and Charm++ runtime software (below) [17]

interaction modules (PPIMs) and programmable geometry cores. To effectively organize these computing units, Anton also customizes on-chip interconnect systems and inter-chip interconnect systems. Unlike classic server chips and AI chips, Anton focuses on low communication latency for better scalability. As shown in Figure 11, the cross-node end-to-end communication latency of the Anton system can reach as low as 90.1 ns. To achieve load balancing in molecular dynamics simulations, Anton also designs a task scheduler and an event-driven programming model that integrates hardware and software.

4 Spatial Computing Design Scheme

As described in Section 2, against a backdrop of increasingly complex application algorithms and hardware resulting from chiplet and wafer-scale technologies, we try to deploy *complex computing tasks* on *large-scale hardware* efficiently by adding an abstraction layer to the computational graph. The scheme described in this paper therefore focuses on applications that require high scalability and scenarios that have complex distribution of hardware resources in the system. For example, a distributed training cluster and an HPC cluster oriented to an AI foundation

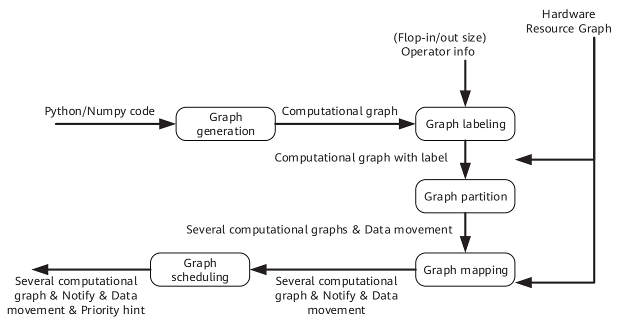

Hardware Resource Graph
(Flop-in/out size) Operator info
Python/Numpy code
Graph generation
Computational graph
Graph labeling
Computational graph with label
Graph partition
Several computational graphs & Data movement
Graph mapping
Graph scheduling
Several computational graph & Notify & Data movement
Several computational graph & Notify & Data movement & Priority hint

Figure 12 Spatial computing flowchart: offline scheduling

network generally integrate a plurality of heterogeneous computing resources. These clusters consist of networks with different topologies across the pod, node, chip, and die levels. In addition, the system includes multiple storage resources with different bandwidths and delays, such as the on-chip static random-access memory (SRAM), in-package high bandwidth memory (HBM), and out-of-package dynamic random-access memory (DRAM). The design scheme of spatial computing is shown in Figure 12. Spatial computing supports the description of algorithms in high-level languages and compilation of algorithms into computational graphs. The deployment and scheduling of computational graphs are classified into two phases: offline scheduling and online running scheduling. These are described in the following sections.

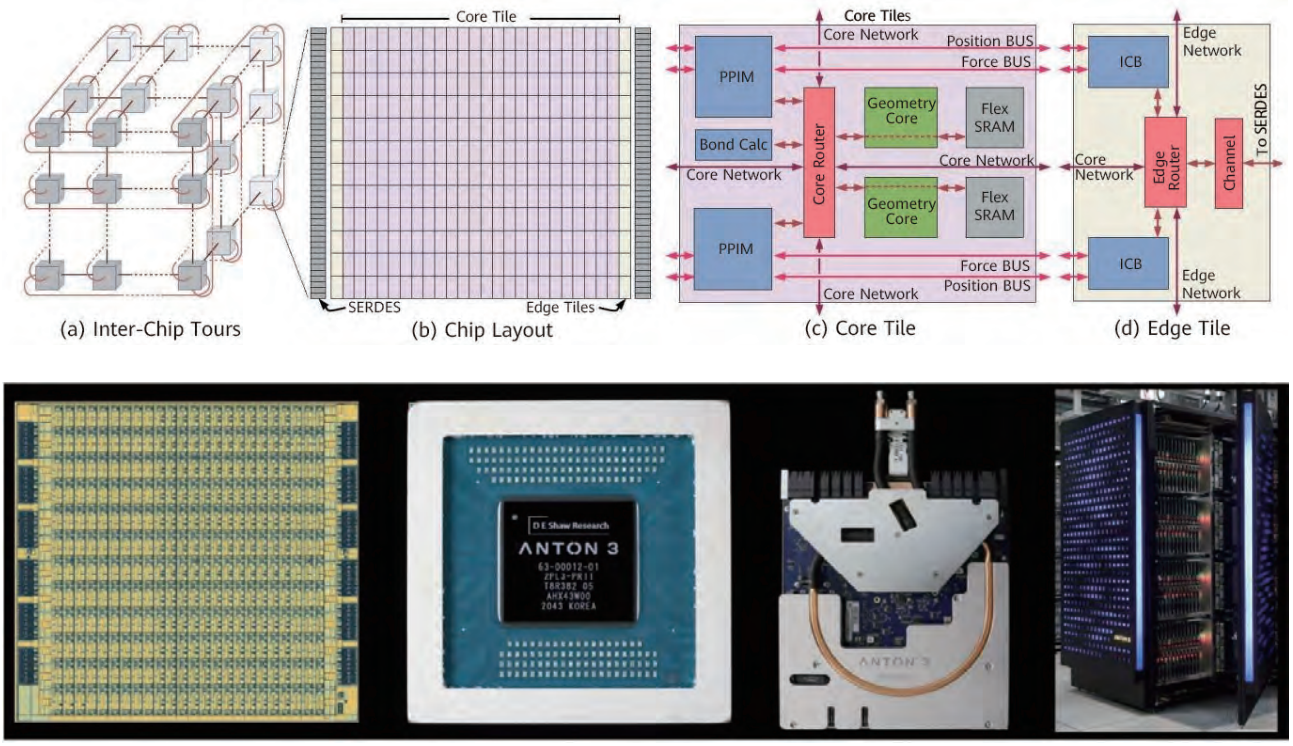


Figure 11 Anton system architecture (above) and Anton photos (below) [18]

4.1 Computational Graph and Hardware Resource Graph

The existing computational graph schemes (such as Legate and Charm++) in the HPC field usually assume high homogeneity of hardware systems. As such, they focus only on the partition, deployment, and scheduling of computational graphs. For example, Fugaku uses homogeneous A64FX compute nodes to achieve 6D torus interconnect. Compute nodes in each dimension are homogeneous and have the same interconnection bandwidth [19]. Similarly, although the Summit system consists of CPU-GPU heterogeneous nodes, the nodes are interconnected through fat-trees to ensure the same interconnect bandwidth among nodes [20].

Conversely, the emerging chiplet/wafer-scale system is heterogeneous. For example, in the DoJo system, the node at the edge of the wafer has far lower memory access latency (due to its converged interconnection bandwidth) compared with the node at the center of the wafer. In addition, in modern computing systems, data movements consume much more time and energy than computing does. Therefore, the deployment of computing tasks requires there be location awareness in order to ensure good hardware utilization. As shown in Figure 12, the input of the spatial computing scheme needs to include a computational graph that describes the load information of the computing task, the operator information of the computing task, and a hardware resource graph that describes the distribution of hardware resources. The computational graph is compiled by high-level languages such as Python and Julia. The operator information includes the size of the input and output data of the computing task and the load of the computing task, as shown in Figure 13. The hardware resource graph includes the computing power of the computing unit, capacity of the memory unit, interconnect topology, bandwidth, and delay, as shown in Figure 13.

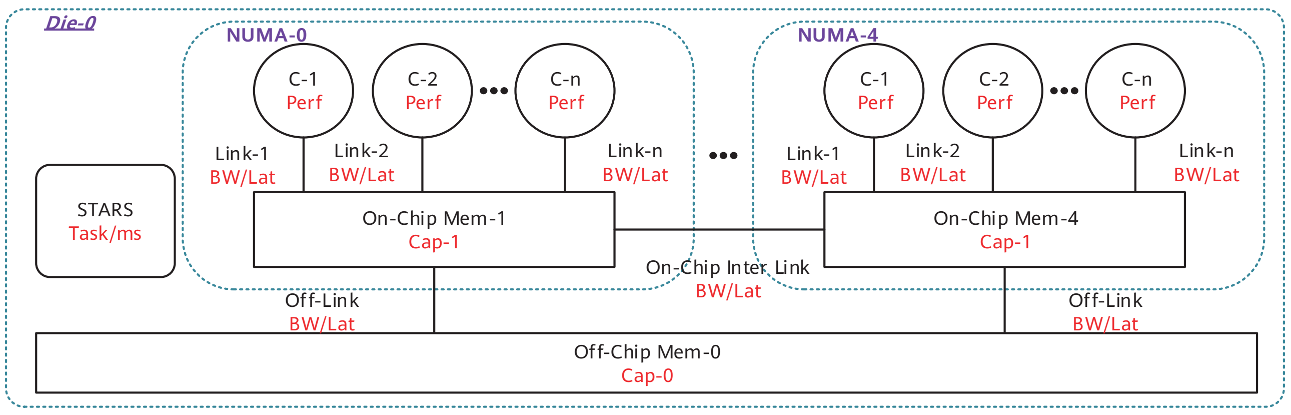
4.2 Annotation, Partition, Mapping, and Pre-scheduling of Offline Computational Graphs

The offline scheduling process based on the computational graph, operator information, and hardware resource graph is as follows:

- **Computational graph annotation**: The input computational graph and operator information are shown in Figure 12. In the computational graph marking stage, the operator information is marked in the computational graph. In Figure 14b, the computational graph contains the unique IDs of all computing tasks, input and output data information of computing tasks, and control flow dependencies between computing tasks. After the marking of the computational graph, each node in the computational graph is added with the load size and required storage size of each computing task. Operator information such as data movement across tasks is added to each edge of the computational graph.

- Operator-1: Flop = f1, Input-Data-Size = {i11, i12, i13}, Output-Data-Size = {o11, o12}
- Operator-2: Flop = f2, Input-Data-Size = {i21, i22, i23}, Output-Data-Size = {o21}
- Operator-3: Flop = f3, Input-Data-Size = {i31, i32, i33}, Output-Data-Size = {o31}
- ......

(a)



(b)

Figure 13 (a) Example of operator information; (b) Example of a hardware resource graph

- **Computational graph partition**: Figure 14c gives an example. The partitions of the computational graph include the information related to the computing tasks (as provided by the marked computational graph) and the hardware resource graph (which includes processor computing power, memory capacity, interconnect latency, and bandwidth). Graph partition is based on the following assumptions:
  (1) The partition of computing tasks is proportional to the computing power provided by the hardware.
  (2) The memory required by the computing tasks is within the memory capacity.
  (3) The cutting edges of the graph are minimized to reduce memory movements.
- **Computational graph mapping**: Figure 15a–b gives an example. The mapping process includes the phases of mapping and communication path planning. In the mapping phase, according to the spatial position relationship between hardware computing units, a method similar to the placement algorithm in the electronic design automation (EDA) is adopted to allocate the sub-graphs divided from the computational graph to computing units with different computing power in different locations. In this mapping process, two tasks involving the transfer of a large block of data are allocated to physically closer hardware computing units. In the communication path planning phase, a method similar to the routing algorithm in the EDA is used. The interconnect bandwidth and topology relationship between compute nodes are considered, and an appropriate path is selected for each data movement task to ensure balanced communication traffic on the interconnect.
- **Computational graph pre-scheduling**: Figure 15c shows a schematic diagram. After the original computational graph is partitioned and mapped, multiple tasks can still be executed at the same time. In the pre-scheduling phase, tasks in the computational graph are sorted by topology in the time domain and executed by priority (PRTY) to optimize the overall task execution efficiency.

Figure 16 shows the offline scheduling result. Each computing unit is assigned a computational subgraph, which contains a series of computing tasks with clearly-declared dependencies, and each computing task is annotated with a priority (PRTY) for reference by the runtime scheduling software. In addition, each subgraph further includes a data movement task across computing units, and a notification task for releasing a control dependency relationship to a remote computing unit.

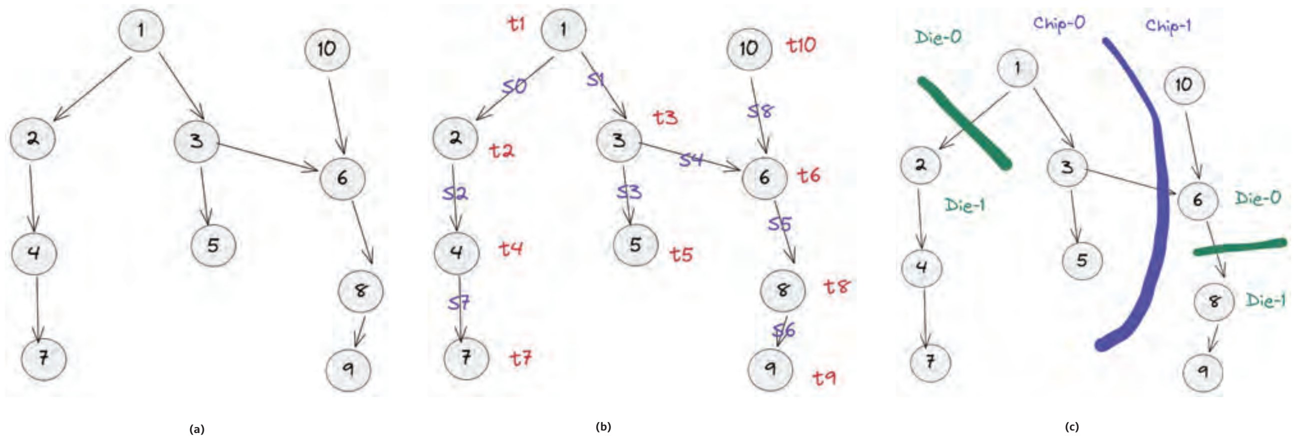


Figure 14 (a) Graph example; (b) Graph annotation; (c) Graph partition

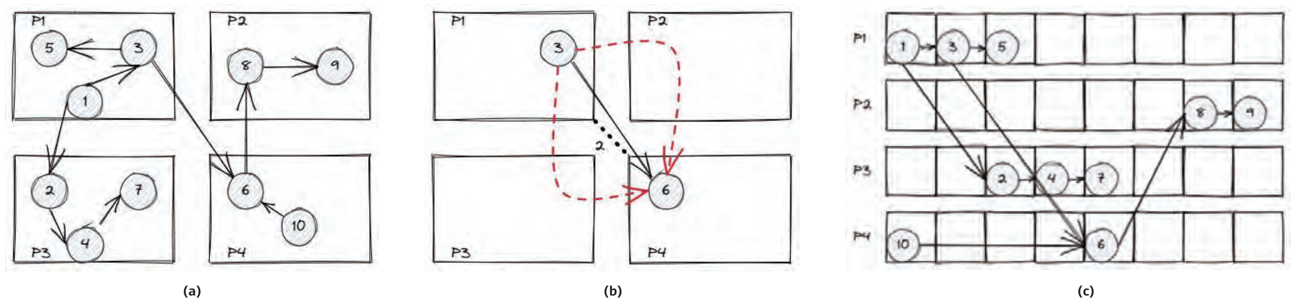


Figure 15 (a) Graph mapping; (b) Graph communication planning; (c) Graph pre-scheduling

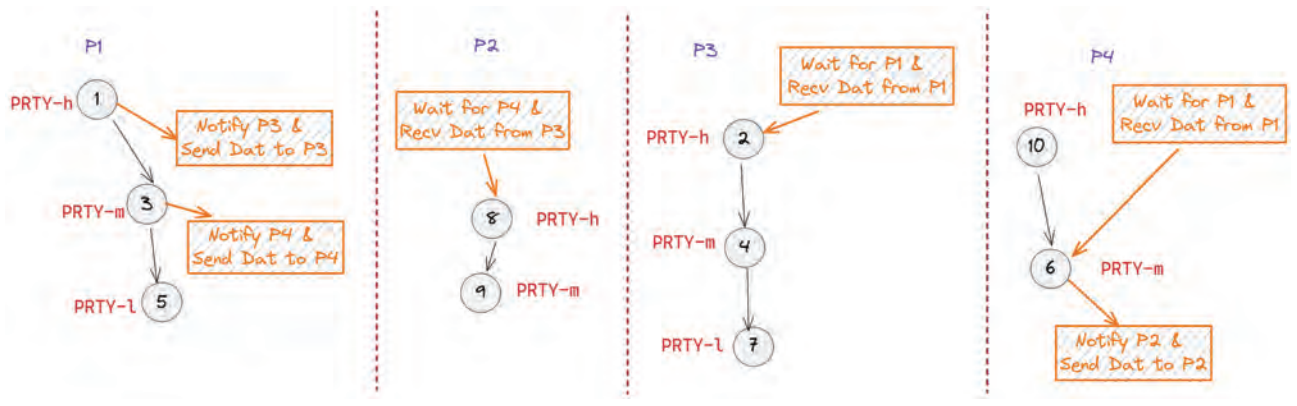

P1
PRTY-h 1
Notify P3 & Send Dat to P3
PRTY-m 3
Notify P4 & Send Dat to P4
PRTY-l 5
P2
Wait for P4 & Recv Dat from P3
8 PRTY-h
9 PRTY-m
P3
Wait for P1 & Recv Dat from P1
PRTY-h 2
PRTY-m 4
PRTY-l 7
P4
PRTY-h 10
Wait for P1 & Recv Dat from P1
6 PRTY-m
Notify P2 & Send Dat to P2

Figure 16 Offline scheduling result

4.3 Runtime Load Balancing and Computational Graph Dynamic Migration

Figure 17 shows the basic modules and functions of the runtime software. According to the aforementioned offline scheduling results, runtime software of each compute node stores a corresponding computational subgraph and maintains a released task queue, an input queue, and an output queue.

In the computing process, the main functions of the runtime software are as follows:

(1) Check the computing task dependency in the computational graph and add those that meet the dependency to the released task queue.

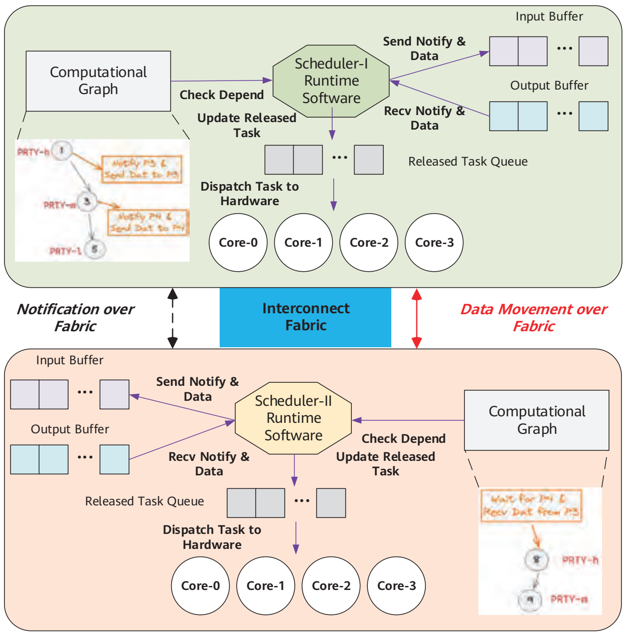


Figure 17 Schematic diagram of the running software

(2) Select a computing task from the release task queue according to the occupation status of the hardware resources and deploy the computing task to the processor core for execution.

(3) When the computing task is completed, if, for example, a subsequent task or input data of the computing task is located at a remote node, send the corresponding notification or initiate a data movement task to the output queue.

(4) Receive notification and data movement tasks in the input queue, and update related information in the local computational graph.

In addition, the runtime software monitors the release task queue usage of the local node and its adjacent nodes. When the occupation of the release task queue exceeds or falls below a certain threshold, the task is migrated in work-stealing or spilling mode. This mechanism can ensure better hardware usage for applications with strong dynamic features, such as sparse matrix solution and molecular dynamics simulation.

5 Prototype System Verification

5.1 Experiment Settings

Based on the spatial computing design scheme mentioned earlier, we build a prototype system to verify its feasibility. The experiment uses the training of an AI foundation network (Megatron network) as an example application. In Figure 18a, the Megatron network includes 72 transformer encoder layers, and each layer in the computational graph includes forward propagation (FP), backward propagation (BP), and gradient descent computing tasks. It is assumed that the chip is comprised of multiple chiplets, including the CPU Die for general-purpose computing and the AI Die for AI computing, as shown in Figure 18b. The hardware resource distribution of the chip is described in the hardware resource diagram in Section 4. The training process of the AI network is similar to the pipeline parallel mode [20] adopted by Microsoft's PipeDream. In Figure 18c, the parallel method is implemented among chiplets at a micro-batch granularity. (Each chiplet computes some of the layers, and processes FP and BP computation in turns based on the dependency.)

5.2 Experiment Results

Figure 19 compares manual deployment and automatic deployment based on the spatial computing scheme mentioned earlier. Manual deployment evenly allocates different layers of an AI network into multiple chiplets, and implements pipeline parallelism similar to that in Figure 18c through programming. This method relies on the programmer's experience to complete the task partition and deployment, and requires manual optimization of the distributed system. Conversely, in the spatial computing scheme, the programmer only needs to use high-level languages to describe the algorithm and adopt the serial programming mode without perceiving the distribution among chiplets. A compiler generates computational graphs and automatically implements partition, deployment, and global optimization with tools. In comparing Figure 19a and Figure 19b, it can be seen that the behavior of the automatic deployment is slightly different from that of the manual deployment. The advantages of the former are as follows:

- The automatic deployment scheme based on computational graphs can simplify the optimization work of programmers. The results of automatic deployment may be slightly lower than those of manual optimization. However, automatic deployment can quickly achieve better performance in new applications or portation of existing applications to a new hardware platform.
- The automatic deployment scheme based on computational graphs can be globally optimized based on computational graphs and hardware resource graphs.
- Based on more comprehensive information and graph optimization algorithms, automated schemes for spatial computing may acquire better deployment strategies.

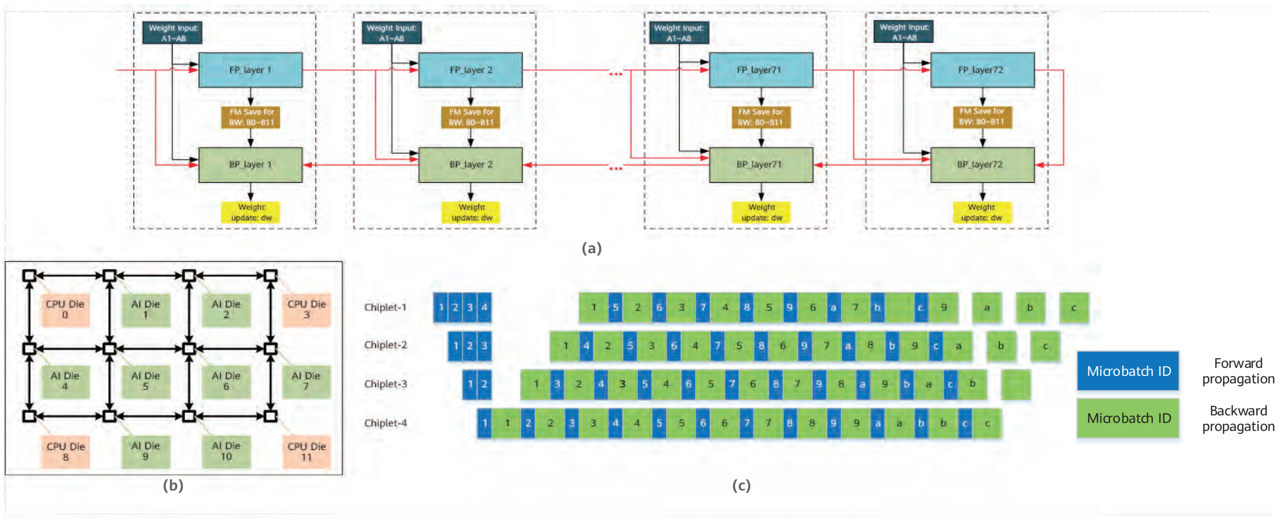
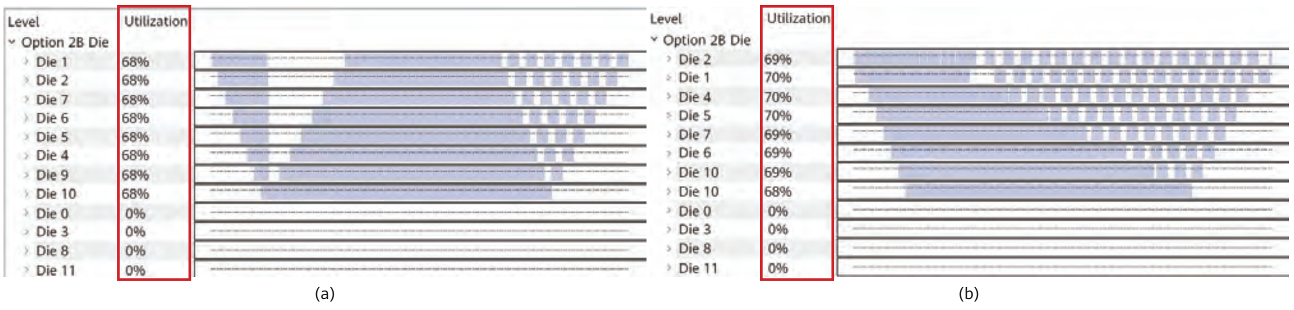


Figure 18 (a) Computational graph; (b) Hardware structure; (c) Scheduling actions



Level	Utilization
Option 2B Die	
Die 1	68%
Die 2	68%
Die 7	68%
Die 6	68%
Die 5	68%
Die 4	68%
Die 9	68%
Die 10	68%
Die 0	0%
Die 3	0%
Die 8	0%
Die 11	0%

(a)

Level	Utilization
Option 2B Die	
Die 2	69%
Die 1	70%
Die 4	70%
Die 5	70%
Die 7	69%
Die 6	69%
Die 10	69%
Die 10	68%
Die 0	0%
Die 3	0%
Die 8	0%
Die 11	0%

(b)

Figure 19 (a) Manual deployment; (b) Automatic deployment

Table 1 Performance comparison between manual deployment and automatic deployment

	Manual Deployment	Spatial Auto-Deployment
ResNet-50	x1	x1.01
Bert-Large	x1	x1.02
VGG-16	x1	x0.98
Inception	x1	x1.01

- The automatic deployment scheme based on computational graphs can add information at high priority for each task in the entire graph. The information can help the runtime software in dynamic load balancing.

Experiment results show that automatic deployment can achieve similar performance to that in manual optimization (2% higher by automatic deployment).

We have also performed similar pipeline parallel experiments on ResNet, Bert-Large, VGG-16, and Inception networks. Table 1 shows that automatic deployment based on spatial computing can achieve performance close to manual deployment.

6 Conclusion

Algorithms represented by the AI method are strikingly complex. At the same time, the growth of the computing power density is slowing due to the slowdown of Moore's Law. Against this backdrop, spatial computing is dedicated to extending computing tasks to a larger spatial extent at the architectural level. However, the methods to achieve high scalability of computation usually rely on professional programmers carrying out manual optimization, resulting in programming difficulties and poor performance portability. Spatial computing tries to add an abstraction layer of computational graphs, take operator-level large-granularity tasks as basic scheduling units, and achieve a tradeoff between programmability and scalability during the graph building and operator building phases.

With numerous attempts, the industry has made remarkable achievements. Based on the industry progress, service requirements, and technology experience, the HiSilicon research team has organized and proposed the preliminary scheme of spatial computing and made some progress. However, the team still faces a series of challenges. For example, the boundary division between computational graph and operator is empirical, meaning that the selection of operator granularity needs systematic methodology. More attempts should be made in high-scalability hardware interconnect architecture for spatial computing. Graph optimization and scheduling algorithms involved in the estimation of operator information and the compilation of computational graphs also involve more algorithmic obstacles.

References

[1] ARDEN W, BRILLOUT M, COGEZ P, *et al*. More-than-moore white paper[J]. Version, 2010, 2: 14.

[2] DAVID E A, L. Molecular representations in ai-driven drug discovery: a review and practical guide[J]. Journal of Cheminformatics, 2020: 1–22.

[3] SMIRNOV N N. Supercomputing and artificial intelligence for ensuring safety of space flights [J]. Acta Astronautica, 2020: 576–579.

[4] HAUPT S E, *et al*. Machine learning for applied weather prediction[C]//IEEE 14th international conference on e-science (e-Science). 2018: 276–277.

[5] LABOMBARD B E A. Evidence for electromagnetic fluid drift turbulence controlling the edge plasma state in the alcator c-mod tokamak[J]. Nuclear fusion, 2005: 12.

[6] News[J/OL]. TOP500, 2017. https://www.top500.org/news/top500-meanderings-sluggish-performance-growth-may-portend-slowing-hpc-market/.

[7] News[J/OL]. OpenAI, 2020. https://openai.com/blog/ai-and-compute/.

[8] TURING A M. On computable numbers, with an application to the entscheidungsproblem[J]. J. of Math 58, 1936: 345–363.

[9] News[J/OL]. Cerebras, 2022. https://www.cerebras.net/product-chip/.

[10] News[J/OL]. SemiAnalysis, 2021. https://semianalysis.substack.com/p/the-tesla-dojo-chip-is-impressive.

[11] ARM. AMBA 5 CHI architecture specification[R]. ARM, 2014.

[12] OWENS SCOTT E A. A better x86 memory model: x86-tso[C]//International Conference on Theorem Proving in Higher Order Logics. 2009.

[13] ACUN B, BUCH R, KALE L V, *et al*. Namd: Scalable molecular dynamics based on the charm++ parallel runtime system[J]. Exascale Scientific Applications: Scalability and Performance Portability, 2017: 119–143.

[14] SZE V E A. Hardware for machine learning: Challenges and opportunities[C]//IEEE Custom Integrated Circuits Conference (CICC). 2017.

[15] SPINELLIS D. Another level of indirection. beautiful code: leading programmers explain how they think[M]. O'Reilly and Associates, 2007.

[16] BAUER M, GARLAND M. Legate numpy: Accelerated and distributed array computing[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2019: 1–23.

[17] KALE L V, KRISHNAN S. Charm++ a portable concurrent object oriented system based on c++[C]//Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications. 1993: 91–108.

[18] SHAW D E, ADAMS P J, AZARIA A, *et al*. Anton 3: twenty microseconds of molecular dynamics simulation before lunch[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2021: 1–11.

[19] SATO M. The supercomputer fugaku and arm-sve enabled a64fx processor for energy-efficiency and sustained application performance[C]//19th International Symposium on Parallel and Distributed Computing (ISPDC). 2020.

[20] HINES J. Stepping up to summit[J]. Computing in science engineering, 2018: 78–82.

Reprioritizing Speculative Task-Level Parallelism

Gilead Posluns [1], Yan Zhu [1], Guowei Zhang [2,*], Heng Liao [2], Mark C. Jeffrey [1]
[1] University of Toronto
[2] Huawei Technologies Co., Ltd.

Abstract

Task-based programming and execution models expose massive parallelism to achieve performance improvement and energy savings. While many implementations are non-speculative, task-level speculation has proved critical when exploiting parallelism in irregular algorithms, such as graph analytics. However, prior models lack expressive semantics regarding scheduler data structures, leaving much performance unexploited.

Specifically, many algorithms schedule tasks, according to a priority order for correctness or faster convergence. While priority schedulers commonly implement task *enqueue* and *dequeueMin* operations, some algorithms need a *priority update* operation that alters the scheduling metadata for a task. Prior software and hardware systems that support scheduling with priority updates compromise on parallelism, work-efficiency, or even correctness.

We present Hive, a task-based execution model and multicore architecture that extracts abundant fine-grain parallelism from algorithms with priority updates, while retaining their strict priority schedules. Like prior hardware systems for ordered parallelism, Hive uses data- and control-dependence speculation and a large speculative window to execute tasks in parallel and out-of-order. Hive improves on prior work by directly supporting updates in the interface, identifying the novel *scheduler-carried dependence*, and speculating on such dependences with task versioning, distinct from data versioning. Hive enables safe speculative updates to the schedule and prevents spurious conflicts among tasks to better utilize speculation tracking resources and efficiently uncover more parallelism. Across a suite of nine benchmarks, Hive improves performance at 256 cores by up to 2.8× over the next best hardware solution, and even more over software-only parallel schedulers.

* Corresponding author

1 Introduction

Task-based programming and execution models have received increasing attention as they are promising to provide better scalability than their conventional *thread-based* counterparts. Multicores strive to keep functional units busy by exploiting parallelism both across threads (thread-level parallelism) and within each thread (instruction-level parallelism). However, thread-based programming models have significant limitations, affecting performance and energy efficiency.

Specifically, thread-based models incur three issues. First, threads are not designed for expressing fine-grain parallelism. For instance, a piece of code with an execution time of ten microseconds can hardly be treated as a thread, because the scheduling overheads in the OS kernel would dominate, causing significant performance loss. Therefore, microsecond-level parallelism can hardly be captured by the thread model. Second, thread-based models are not as expressive and rely on OS schedulers heavily. A typical practice of thread programming is to create many threads, often many more than the hardware can support concurrently, to use synchronization primitives to express their dependencies separately, and to leave them to the OS scheduler [49]. Third, many *irregular* applications, where control flow and data accesses are hard to know statically, still suffer from limited available parallelism, e.g., they are hard to scale to a few tens of cores or hard to program [31].

Task-based models are much more surgical than thread-based ones. First, tasks can easily carry additional information, such as dependences or timestamps, to express complex control or data flows. For instance, OpenMP 4.0 [13] allows programmers to specify the data dependences of a task. Swarm [25, 26] allows users to use per-task timestamps to enforce a task order. Second, task-based models typically adopt a two-tier scheduling mode. While thread-based models have kernel-level scheduling only, task-based models adopt both kernel-level scheduling, which maps threads to hardware execution units, and user-level scheduling, which maps tasks to threads. This hierarchical design reduces the burden on the OS scheduler, and is especially useful when the system workloads are high. Third, this user-level scheduling is a natural fit for hardware acceleration. The hardware-assisted timestamp-based conflict detection and resolution in Swarm are great examples. Such architectural support can significantly reduce scheduling overheads, enabling more fine-grain parallelism to be exploited, and hence resulting in better scalability and energy efficiency.

Prior work on task-based models can be categorized based on whether it uses speculation to extract task-level parallelism. Though instruction-level speculation is fundamental for modern out-of-order processors to extract instruction-level parallelism, task-level speculation has caused debates, as it has demonstrated great performance potential in some cases, but may incur high area overheads and design complexity.

Specifically, non-speculative task-based programming models have been widely adopted. Many programming languages are nonspeculative, such as OpenMP [13], NESL [12], Cilk [19], and X10 [14]. Such systems avoid the complexity brought in by speculation, such as version management and conflict detection, but leave performance on the table.

Speculative approaches have been studied extensively in academia, mainly through three dominant execution models. Thread-level speculation (TLS) automatically extracts task-level parallelism from sequential programs. Transactional memory (TM) performs optimistic synchronization of usually unordered transactions in multithreaded programs. Swarm adopts timestamps in the programming model to express and force the order of tasks, and relies on architectural support to exploit the parallelism across small independent tasks.

However, prior ordered speculative task-based models still suffer from limited semantics. Typically, they must add software scheduling structures that shadow the hardware ones, tracking much of the same state redundantly, and they must write early exiting tasks to emulate the sequential behavior. Such programs clog the speculative task-tracking data structures of the hardware, resulting in stalls and reduced throughput. In this paper, we focus on a specific operation: *priority updates*.

2 Motivation

2.1 Priority Updates

The optimal or fast-converging algorithms for many problems require their work items, or tasks, to execute according to some priority order. Sequential implementations use a priority queue to schedule the tasks.

In particular, we focus on those algorithms that dynamically alter the task schedule with *priority update* operations [18, 58] that associate an object ID with a priority. For example, some graph algorithms will assign an initial priority to every vertex (ID). Their executions consist of processing the highest-priority vertex, updating the priorities of its neighbors, and then repeating the process in a loop with the next highest priority vertex.

Ideally, these algorithms should have abundant task-level parallelism, as true data dependences among tasks (loop iterations) are rare for sparse data structures like graphs. However, practically extracting this parallelism is challenging as it requires *(i)* ensuring irregular data dependences flow in the required order and *(ii)* circumventing the false data dependences on the global scheduling structure, which every task would otherwise read and write. Software and hardware systems exploit this ordered irregular parallelism [41] using one or more of three techniques: bulk-synchronous parallelism, speculative parallelism, or relaxing the order.

Current software parallel frameworks strive to drive down scheduling overheads. Bucketing [16, 65] is a bulk-synchronous approach that executes groups of equal-priority tasks (buckets) in parallel.

Bucketing can retain a strict priority order, giving work-efficient implementations, but the barriers between buckets limit parallelism when there is little work per bucket. Moreover, priority updates on the schedule can create even more buckets, further constraining parallelism. Speculative techniques [11, 22, 23, 30] uncover parallelism *across priorities*, speculating that tasks will access independent data, circumventing barriers. However, speculation overheads in software overwhelm the benefits of inter-priority parallelism for small tasks [22, 23]. Schedulers that *relax* the priority order [4, 5, 32, 39, 44, 48, 51, 61, 63, 66] present a middle ground between these techniques, providing a best effort to dispatch tasks in order, but with only probabilistic guarantees, if any. Approximating the desired task order circumvents barriers and enables distributing software queues to reduce contention due to scheduling. However, relaxation is only amenable to those algorithms where task ordering is not required for correctness, but instead reduces redundant work to improve convergence time [2, 3, 36, 39]. Moreover, the higher the core count, the greater the deviation from the desired priority order, worsening work efficiency and convergence time [6, 51].

Prior order-aware hardware systems are subject to the same parallelism vs. relaxation trade-off, or do not support priority update operations. PolyGraph [15] provides relaxed priority semantics or accelerated bulk-synchronous execution, but does not provide a scalable strict priority schedule with update semantics. Thread-level speculation [21, 28, 46, 47, 54, 55, 57] targets the automatic parallelization of sequential code, so it couples loop iteration order to execution order. Consequently, to schedule new work to execute at a future time, a scheduling structure in software is still required, serializing all tasks with dependences through the scheduler. In contrast, hardware for speculative ordered parallelism, such as Swarm [25, 26], Fractal [56], and Chronos [1], can implement a dynamic strict priority schedule. However, the sequential queue that these systems abstract supports only *enqueue* and *dequeueMin* operations, notably not a priority update.

Hive [43] is an execution model and speculative multicore architecture to express and extract parallelism from ordered algorithms with priority updates. We characterize the implications of a strict priority schedule with updates, and the complex *scheduler-carried dependences* created between successive tasks in the schedule. The Hive execution model enables the programmer to convey the desired priority schedule (and updates) directly to hardware, abstracting a strict priority queue. Our Hive implementation adds modest area to the Swarm architecture, extracting parallelism from the abstract queue by speculatively executing tasks out of priority order. Importantly, Hive introduces *task versioning*, a method of speculating on scheduler dependences, similarly to how memory versioning enables data dependence speculation.

2.2 Case Study: *k*-Core Decomposition Problem

The optimal algorithm for the *k*-core decomposition problem [35, 50] (`kcore`) requires a strict priority schedule supporting priority updates, and illustrates the challenges and opportunities in this work. The maximum core, or *coreness*, of a vertex is an important property in a variety of domains, including graph mining [52], graph visualization [8], statistical mechanics [37], ecosystem analysis [38], and bioinformatics [62]. The coreness can represent the importance of a vertex [34] or its position in a hierarchical description of the graph [8]. A *k*-core of an undirected graph is a maximal set of vertices where every vertex has at least *k* edges to other vertices in the *k*-core.

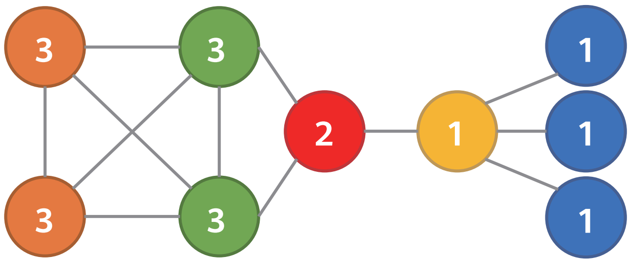


Figure 1 Graph labeled with the coreness of each vertex

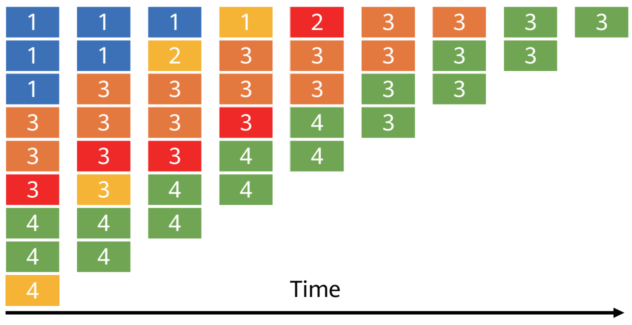


Figure 2 `kcore` priority queue contents (A vertex is dequeued from the top at each time.)

Listing 1 shows the sequential algorithm for `kcore`, which determines the coreness of each vertex in the graph. In essence, for each increasing value of *k*, the algorithm recursively removes all vertices with degree *k*, then repeats with the next value of *k*, until no vertices remain. The *k* value at which a vertex is removed is its *coreness*.

```
PriorityQueue pq;
for (int v : G.V)
  pq.inquire(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

Listing 1 Sequential code for `kcore`

This algorithm depends on three priority queue operations. In addition to the `enqueue` and `dequeueMin` operations typical of any priority queue [60] (e.g., the `C++std::priority_queue`), it also uses `decrementPrio` [18, 58] (e.g., in the `boostfibonacci_heap`).[1] The latter operation indexes into the queue by vertex (ID) and updates its priority to dynamically alter the vertex's position in the schedule.

[1] This code adds a fourth `getPrio` command for readability.

The priority schedule of `kcore` is required for correctness. However, there is ample parallelism available in `kcore`, once the false data dependences on the scheduling structure [25] are abstracted away. In the running example, iterations that operate on the blue and orange vertices are independent and could therefore be processed in parallel. However, current software and hardware are incapable of efficiently unlocking this parallelism for the following reasons.

Priority updates often outnumber dequeues: Since vertices on average have more than one neighbor, then `kcore` will at least consider calling `decrementPrio` more than it calls `dequeueMin`. Figure 3 shows the high ratio of conditional updates vs. dequeues on large inputs for our benchmark suite of nine algorithms, including `kcore` (see Section 6.1 for methodology). The ratio is greater than 1 for all benchmarks except `astar`, which terminates before exploring the entire graph, and `mm`, where priority updates can only eliminate two thirds of the fine-grain tasks [24], capping the ratio at 1. Since priority updates can comprise the majority of an algorithm's work, they must be performed in a scalable and efficient way.

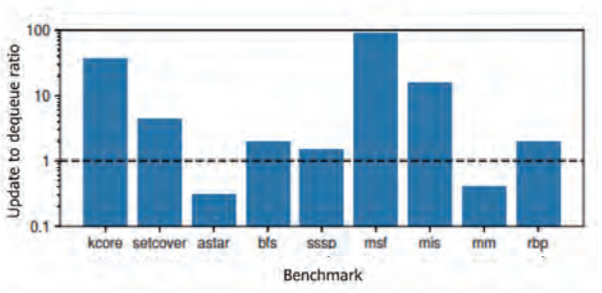


Figure 3 Ratio of scheduler updates to dequeues at 1 core

Priority updates cause poor performance in software: Like `enqueue` and `dequeueMin`, priority updates are read-modify-write operations on the scheduling structure, so they contend when performed on a shared global state. Software schedulers with privatization [32, 39, 51, 63] mitigate some contention, but they consequently do not update the global scheduler immediately. Therefore, to maintain `kcore`'s strict priority schedule, systems such as Julienne [16] and Ordered GraphIt [65] use bulk-synchronous parallelism among equal-priority tasks and apply reductions to the privatized queues into a consistent state at barriers. These systems can only extract parallelism from the potentially limited work between barriers, and are unable to extract parallelism across priorities.

To extract parallelism across priorities, the alternatives to synchronous execution are relaxation, speculation, and task dependence graphs. Relaxed priority queues are ineligible for `kcore` because relaxed scheduling can lead to incorrect outputs. `kcore`'s tiny tasks and abundant updates would lead to high overheads in scheduler-aware speculation [22, 30], and kinetic dependence graphs [23], as observed in similar algorithms.

Priority updates have poor performance in hardware: Hardware implementations do not support explicit priority update operations without either relaxing the schedule [15] or requiring schedule tracking metadata in software.

Swarm [25] and Chronos [1] provide strict priority scheduling, using hardware speculation to extract parallelism across priorities, but lack built-in support for priority updates. A programmer wishing to write a program with priority updates must (*i*) implement a scheduling metadata structure in software to track the current priority of each object, and (*ii*) restructure task code to check the metadata and exit early if the task's object priority has been updated, making the task moot. This is similar to writing Listing 1 with a priority queue that only supports *enqueue* and *dequeueMin*, as shown in Listing 2. This code is largely similar to Listing 1, with the exceptions of lines 10 and 12–14. Line 10 checks the priority-tracking metadata structure to ensure that vertex `v` was not already processed at an earlier priority. If it was, the loop exits that task early and moves on to the next vertex. Because the queue does not support priority updates, lines 12–14 instead conditionally decrement the priority of `v`'s neighbors in the scheduling metadata and enqueue a new task for each vertex `nbr` at

```
 1 PriorityQueue pq;
 2 int prios[G.n]; // Scheduling metadata
 3 for (int v : G.V) {
 4   prios[v] = G.degree[v];
 5   pq.enqueue(v, prios[v]);
 6 }
 7 while (!pq.empty()) {
 8   int v, int prio = pq.dequeueMin();
 9   // Skip if this iteration/task is moot
10   if (prio > prios[v]) continue;
11   for (int nbr : G.edges[v])
12     if (prios[nbr] > prio) {
13       prios[nbr]--;
14       pq.enqueue(nbr, prios[nbr]);
15     }
16 }
17 coreness = prios
```

Listing 2 Sequential `kcore` without priority updates

its new priority. When the old later-ordered task dequeues, it will exit early at line 10.

Having tasks check a condition and potentially do nothing is similar to predication of instructions as an alternative to conditional branches [7]. Early exiting tasks fill the speculation state-tracking structures with tasks that are practically NOPs. Although task-level predication is a valid approach, we advocate for a more expressive execution model and hardware support to better utilize on-chip resources for extracting reprioritizable ordered parallelism.

3 Hive Execution Model

A Hive program consists of priority-ordered *tasks* which can be logically bound to *objects* to enable updates to the schedule. Hive hardware extracts speculative parallelism across hundreds of cores by finding independent tasks to run out-of-order. However, Hive guarantees the program output will always match that of a sequential thread scheduling the tasks in a priority queue supporting updates [18, 58]. Every task can read and write arbitrary shared memory, dynamically *update* tasks bound to objects, and enqueue new tasks unbounded from any object. The update and enqueue operations assign each task an integer *timestamp* which encodes its priority order in the modeled queue. Hive objects are the program's core data type for scheduling, and each object is identifiable with a unique number, such as a memory address or ID. The programmer defines objects as needed in application memory. Hive records the binding for every object ID to one or no queued task in an *object table* residing in a protected region of memory, inaccessible to the tasks except through an API. When no queued task is bound to an object, the object table instead records the timestamp of the object's last queued task (or infinity if none).

Hive ensures that tasks *appear to* execute in increasing timestamp order, as if a sequential loop repeatedly dequeues the lowest-timestamp task from the modeled queue, runs it, then dequeues the next task, until the queue is empty. Tasks with equal timestamp are atomic, being serialized arbitrarily among each other. This execution model has two key consequences: *(i)* a task's enqueued child tasks appear to execute only after the parent task finishes — child tasks are ordered after the parent task — and *(ii)* a task's accesses to shared memory and its updates to object-task binding appear to happen atomically and before the next task in priority order would be dequeued.

Table 1 shows the Hive API, which programs use to enqueue tasks and manipulate the object-task bindings. Listing 3 shows the API in action with a Hive implementation of `kcore`.

A Hive task is an instance of a function with timestamp and arguments received through registers. Listing 3 defines one task function, `removeV`, which logically removes a vertex `v` (a Hive *object*) from the graph, performing the work of lines 6–9 of Listing 1.

Any task can set, `update`, or `cancel` tasks bound to objects, by calling the given (inlined) Hive functions with a task function pointer, timestamp, and arguments. These are passed to hardware through registers with one new instruction. While these *basic* operations provide sufficient schedule manipulation for some programs, others require *timestamp-relative* task updates that depend on the timestamp of the task currently bound to an object. One could implement such relative updates with `hive::getTS` and `hive::update`, but *(i)* this complicates programming, and *(ii)* it hurts performance with remote accesses to the object table in memory. The Hive interface improves expressiveness by adding min, increment, and decrement updates, similar to those in DSLs [65].

Listing 3 uses both basic and relative task updates. The main function binds an initial `removeV` task to every vertex by calling `hive::update` with timestamp equal to `v`'s original degree. The `removeV` task itself decrements the task timestamp (degree) for all neighbors of its vertex `v`. Since `kcore` tasks are ordered by their current degree, `removeV` implicitly sets `v`'s coreness in the object table as the last (and only) timestamp for a task that executed on `v`.

```
void removeV(Vertex* v, Timestamp ts) { // Task
  for (Vertex* nbr : v->neighbors())
    hive::decrTS(&removeV, nbr, 1);
}
void main(int argc, char** argv) {
  hive::init(G.n);
  for (Vertex* v : G.V)
    hive::update(&removeV, v, v->degree);
  hive::run();
  hive::extract(coreness);
}
```

Listing 3 Hive implementation of `kcore`

Table 1 Hive programming interface

Signature			Description
	void	`hive::init<flags>(nobjs)`	Reserve object table capacity for `nobjs` objects with no initially queued tasks. Empty (null) tasks implicitly have timestamp infinity.
	void	`hive::extract(Timestamp* dest)`	Extract the timestamp of the last task that executed for every object ID.
Basic	void	`hive::update(taskFn, oid, ts, args...)`	Replace or set the queued task bound to object `oid`.
	void	`hive::cancel(oid)`	Remove the queued task bound to object `oid`. This is either an `update` with timestamp infinity, or an `updateMin` with timestamp 0 and an empty task.
	Timestamp	`hive::getTS(oid)`	Return the timestamp of the task currently or last executed bound to `oid`.
	void	`hive::enqueue(taskFn, ts, args...)`	Queue a task unbounded from any object.
TS-Relative	void	`hive::updateMin(taskFn, oid, ts, args...)`	Replace the queued task bound to object `oid` only if `ts < hive::getTS(oid)`.
	void	`hive::incrTS(taskFn, oid, delta, args...)`	Replace the queued task bound to object `oid`, adding a signed `delta` to the previous timestamp. NOP if there is no task bound to the object.
	void	`hive::decrTS(taskFn, oid, delta, args...)`	Replace the queued task bound to object `oid`, where the new timestamp is equal to the max of the previous timestamp minus an unsigned `delta` and the caller's timestamp, only if doing so would decrease the timestamp. NOP otherwise, or if there is no task bound to the object.

A program invokes Hive by initializing the object table (`hive::init`), enqueuing or updating some initial task(s), then calling `hive::run`, which returns control to the main thread when there are no more tasks to run. Listing 3 initializes the object table with one entry for every vertex in the graph. In many algorithms, the program output is in fact the priority at which every task ran, so Hive provides the `hive::extract` function to copy the last timestamp for every object to a buffer in application memory.

4 Speculative Parallelism amid Priority Updates

Hive software directly conveys its dynamic scheduling needs to hardware, expressing implicit parallelism through task enqueues and updates. Our goal with the Hive implementation is to extract abundant parallelism among the queued tasks. Because safe parallel executions for these irregular algorithms cannot be statically determined at compile time, Hive executes queued tasks in parallel and out-of-order, speculating that they are independent. Specifically, Hive speculates that for every executing task:

- its data-dependent predecessors have performed their stores,
- its parent task did not misspeculate, and
- it will not become Moot: replaced or canceled.

The Swarm architecture [25] similarly speculates on the first two conditions (data and control dependences, respectively), making it a natural baseline upon which to build Hive. This section gives an overview of Hive's approach to parallelism, identifying similarities and differences between Swarm and Hive, and highlighting the key insights that enable Hive to speculate efficiently amid priority updates.

4.1 Similarities and Differences with Swarm

Both Swarm and Hive detect data misspeculation by tracking the memory read and write sets of all tasks and piggy-backing on cache coherence requests. When two tasks access the same data with at least one writing, both systems identify a *dependence order violation* based on task timestamps and access types, and abort and restart the later-ordered task if necessary.

Both systems handle control misspeculation by tracking child tasks. A parent task may have created its child tasks based on incorrect control flow due to misspeculating on data. If the parent task aborts, both systems abort all its control-misspeculated descendents, recursively.

The key distinction of Hive hardware from Swarm is Hive's built-in support for updates to the schedule, or how it speculates on *scheduler-carried dependences*. While Swarm can only rely on its support for data and control misspeculation, Hive maintains multiple versions of scheduler-dependent tasks, with all but one per object being in a Moot state. This reduces Moot task overheads, while retaining the ability to recover from priority update misspeculation.

4.2 Scheduler Dependences and Moot Tasks

We identify the *scheduler-carried dependence* as a new class which resembles both data and control dependences, but is neither. A task t is *scheduler-dependent* on task s if, when s appears to begin executing, t is scheduled after s, and either

- s mutates scheduler state corresponding to t, or
- t is scheduler-dependent on u, and u is scheduler-dependent on s.

Scheduler-carried dependences arise in systems where scheduler state of future tasks is accessible and mutable. One example is when a Hive task s cancels the task t bound to some object. The execution or existence of t is scheduler-dependent on s. Another example is in processors with self-modifying code [59], if we view each instruction as an individual task. To exit a loop, a store instruction would overwrite the jump target address in the memory location of a later PC [20]. The jump instruction is scheduler-dependent on the store. Swarm tasks are never scheduler-dependent, because Swarm has no mutable scheduler state.

Like how predication can transform control dependences into data dependences [7], one can transform scheduler-carried dependences into a combination of control and data dependences. Consequently, Swarm can implement the Hive execution model using only its data- and control-dependence speculation. For example, Listing 4 shows a Swarm implementation of `kcore`. We replace each Hive operation with reads and writes of scheduling metadata in memory (`prios`) and task `enqueue` (line 7). Now, every update operation produces a new task, and each task first checks memory to see if it is still scheduled to run, exiting early if not (line 3).

```
 1 Timestamp prios[G.n]; // Scheduling metadata
 2 void removeV(Timestamp ts, Vertex* v) { // Task
 3   if (prios[v->id] < ts) return;
 4   for (Vertex* ngh : v->neighbors()) {
 5     if (prios[ngh->id] <= ts) continue;
 6     prios[ngh->id]--;
 7     swarm::enqueue(&removeV, prios[ngh->id], ngh);
 8   }
 9 }
10 void main(int argc, char** argv) {
11   for (Vertex* v : G.V) {
12     prios[v->id] = v->degree;
13     swarm::enqueue(&removeV, prios[v->id], v);
14   }
15   swarm::run();
16   coreness = prios;
17 }
```

Listing 4 The Swarm implementation of `kcore` transforms scheduler-carried dependences into data and control dependences, as did Listing 2 for sequential code.

Priority update operations often outnumber dequeues in algorithms with updatable priority queues. Therefore, the majority of tasks in Swarm implementations will dequeue, perform a single memory read, and exit early with no effect on program state. For example, the early exit path of Listing 4 will trigger more than 36× more often than not (Figure 3). We call these early exiting tasks Moot because they might as well have not run at all.

In a Swarm system, every Moot task consumes cycles on a core while waiting on its memory access, and its speculative state with non-empty read set consumes precious hardware resources until it commits in order. Wasting core and speculation resources on Moot tasks can cause stalls, hurting program performance.

In contrast, Hive speculates on a lack of scheduler-carried dependences among tasks, but it recovers from misspeculation by holding multiple speculative *task versions* for the same object. Unlike Swarm, Hive does not treat these versions as separate tasks: only one will appear to dequeue and run, matching the sequential semantics.

Hive temporarily holds the others in an explicit Moot state. Hive does not execute Moot task versions and aborts a task if it becomes Moot after dequeue. Hive clears Moot versions out of its hardware resources when their fate becomes non-speculative. This is always earlier than they would be committed as normal tasks in Swarm. We provide more detail in Section 5.

5 Hive Implementation

Given the overview of Hive's approach to speculative parallelism, we now present an implementation of Hive as an extension to the Swarm microarchitecture [24, 25, 27], visualized in Figure 4. With restrictions, Hive could also be adapted to the Chronos [1] accelerator for speculative ordered parallelism, which we leave to future work. We first describe Swarm's main features for task-level data and control speculation. We then turn to Hive's modifications that enable detection and recovery from scheduler-dependence misspeculation.

5.1 Hive Microarchitecture

Hive generalizes the Swarm microarchitecture to support *(i)* logically binding ordered tasks and objects, and *(ii)* speculating on the outcome of scheduler-carried dependences. To implement the former, Hive introduces the *object table* in memory. To achieve the latter, Hive adapts Swarm task unit structures, shown in Figure 4, to enable task versioning. For simplicity, we describe Hive hardware *assuming only support for the* `hive::update` *operation*, along with `hive::init` only. The full paper [43] contains the complete description of other operations.

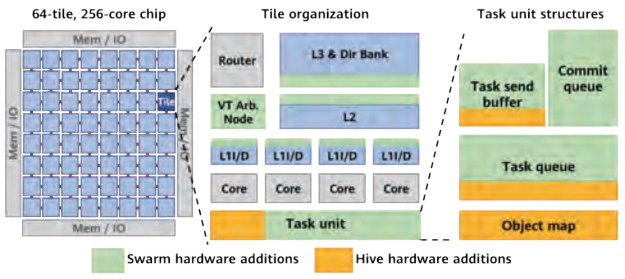


Figure 4 Swarm and Hive system configuration

Object table entries represent non-speculative object-task bindings. The object table has an entry for every Hive object *o*, pointing to the task version currently (or last) bound to *o*. Misspeculating tasks must never corrupt the object table.

The programmer specifies the number of object table entries to allocate with `hive::init`. We implement the object table as an array and leave dynamic creation and destruction of objects to future work (e.g., via hash table or tree).

Speculative task versioning: Hive buffers task descriptors in the TQs for all task versions for all objects, alongside any enqueued tasks unbounded from objects. A task calling update creates a task with a similar asynchronous send scheme as enqueue. Like enqueue, the parent task tracks the location of its child task. Unlike enqueue, the child task may be speculatively replacing (or replaced by) another task for the same object, making it a speculative task version.

Hive sends new task versions for the same object to the same tile. Like Swarm's spatial hints insight, tasks bound to the same object are likely to access the same data, leveraging locality. Unlike (optional) spatial hints, co-locating same-object tasks is required to prevent races on the object table and simplify detection of scheduler-dependence misspeculation.

Mootness detection and recovery: Hive must detect and recover from two cases of scheduler-dependence misspeculation. First, a task version that is idle, running, or finished could be replaced by an update operation, making it speculatively Moot. Second, a task version previously found to be Moot may have been replaced by a misspeculating task calling update, and should be restored.

Hive expands the Swarm task descriptor [25, 26], to facilitate mootness detection. Update operations add the *virtual time* (VT) of the task responsible for creating the task version. This is usually the parent task that directly called update, but read-only spawner trees [64] propagate the parentVT from their root task to increase parallelism.

A task unit detects which task versions are newly made Moot when it receives a new task descriptor for a priority update to object *o*. The task unit inserts the incoming task version *ti* to a free slot in the TQ and object map. and compares two VT fields of *ti* with those of all task versions for o in the TQ. Based on the following rules, moot tasks are determined and actions such as aborts or slot replacement are performed accordingly.

$parentVT(t_q) < parentVT(t_i) < VT(tq) \Rightarrow t_q$ is Moot
$parentVT(t_q) < VT(t_q) < parentVT(t_i) \Rightarrow$ neither is Moot
$parentVT(t_i) < parentVT(t_q) < VT(t_i) \Rightarrow t_i$ is Moot
$parentVT(t_i) < VT(t_i) < parentVT(t_q) \Rightarrow$ neither is Moot

Object map: Hive adds an *object map* to every task unit to accelerate task-version queries. This associative array maps an object ID to the set of TQ entries that hold task versions for the given object. We move the object ID field out of the TQ and into the object map.

Clearing Moot task versions: When a task commits, its child tasks are no longer control-speculative but they may remain data-dependent or scheduler-dependent on other speculative tasks. Swarm and Hive virtualize the TQ by making idle control-non-speculative tasks spillable to memory. Hive also clears Moot tasks from the TQ and object map by exploiting scheduler-non-speculative object-task bindings.

5.2 Hive Overheads

Swarm adds modest overheads [25] to a multicore and Hive adds more, with the key addition of a 2KB CAM for the object map per task unit. We use CACTI 7.0 [10] to estimate the area of Bloom filters, SRAMs, and CAMs for a 32 nm process. We estimate the order queue area by scaling a commercial 28 nm TCAM [9]. The total storage area of a Hive task unit is less than 3% of a 45 nm Nehalem processor [17] when scaled up to a 45 nm process.

6 Evaluation

We evaluate Hive across nine benchmarks that require, or benefit from, priority scheduling with updates. We find that Hive consistently outperforms Swarm and software-only parallel implementations, with speedups of up to 2.8× over Swarm at 256 cores (gmean 52%) and more over software. Papers [42, 43] also characterize Hive and Swarm performance sensitivity to graph structure, and show the performance impact of restricting scheduling features.

6.1 Methodology

Modeled system: We adapt an open-source[1], cycle-level, execution-driven simulator based on Pin [33, 40] to model Hive, Swarm, and multicore systems of up to 256 cores. Swarm parameters are consistent with prior work [24, 25, 27, 56, 64]. We use detailed core, cache, network, and main memory models, and simulate all task and speculation overheads (e.g., task traffic, running misspeculating tasks until they abort, simulating conflict and mootness check and rollback delays and traffic, etc.). We also simulate smaller systems with square meshes ($K \times K$ tiles for $K \leq 8$), keeping per-core cache sizes and queue capacities constant. Because aggregate cache and queue capacity grows, we see superlinear speedup on several benchmarks.

[1] https://github.com/SwarmArch/sim

Benchmarks: We ported four graph benchmarks to Hive (astar, bfs, sssp, and mis) from prior Swarm work [1, 25, 27, 56]. We also ported one statistical inference (rbp [2]), one optimization (setcover [16]), and three graph algorithms to Hive and Swarm: kcore [16], msf (minimum spanning forest) [29, 45, 53] and mm (greedy maximal matching) [53]. The full paper [43] contains a complete description of our methodology.

We fast-forward each benchmark to the start of its parallel region and run the entire parallel region. We perform enough runs to achieve 95% confidence intervals ≤ 1%.

6.2 Hive Performance

Figure 5 compares the performance of Hive, Swarm, and software-only parallel versions of our benchmarks, as the system scales from 1–256 cores. Hive always outperforms both, with speedups over parallel software of 3.3× (rbp) to 124× (sssp) at 256 cores (gmean 23×). Software-only versions struggle to scale to hundreds of cores on these inputs, so we do not consider them further. The benefits of Hive over Swarm vary with algorithm and input. At 256 cores, Hive yields between modest speedups of 11–22% (astar, mm, rbp) to large speedups of 1.9× (sssp) and 2.8× (kcore) (gmean 52%).

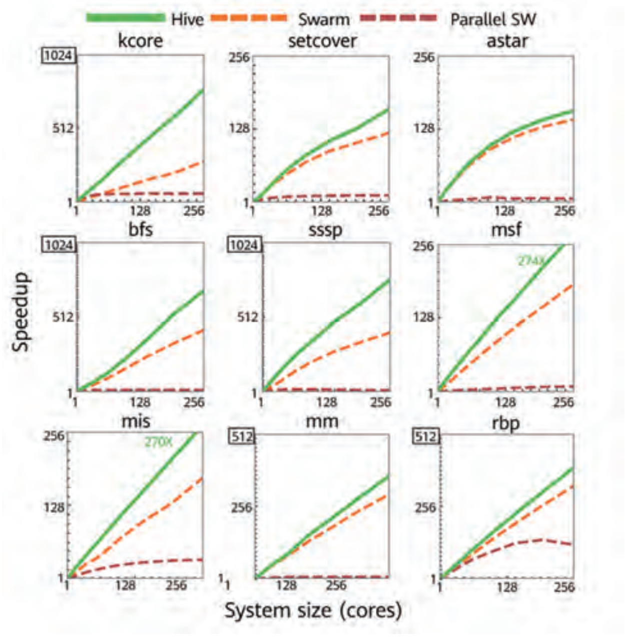


Figure 5 Speedup of Hive, Swarm, and software-only versions on 1–256 cores, normalized to tuned 1c Swarm

Figure 6 gives more insight into these results by showing execution time breakdowns for Swarm and Hive versions at 256 cores. Each pair of bars shows a benchmark, with the height of a bar giving the execution time relative to Swarm. Each bar breaks down how cores spend their cycles, executing *(i)* tasks that eventually commit or *(ii)* later abort; and cycles spent (iii) spilling tasks to/from memory; *(iv)* stalled on a full TSB or CQ; or *(v)* idle because there are no (non-Moot in Hive) tasks available to run. The figure overlays the update-to-dequeue ratio of Figure 3 on the right y-axis. We analyze overall trends first, then focus on outliers kcore and mm. The results show that Hive reduces committed task cycles across all benchmarks, and reduces the cycles stalled on resource exhaustion for most of them.

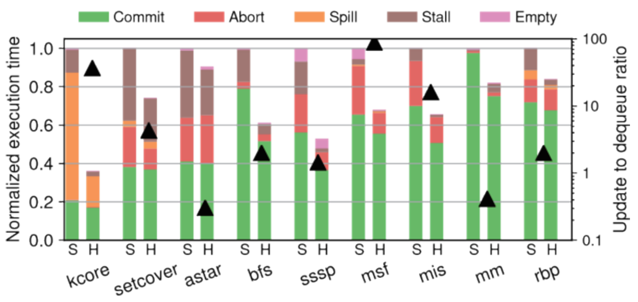


Figure 6 Breakdown of total core cycles at 256 cores, comparing Swarm and Hive (lower is better)

7 Conclusion

Foundational and emerging algorithms depend on a strict task ordering for correctness. However, hardware systems that support task order lack crucial update operations. We have examined the semantics of the priority update operation, and in doing so, uncovered a new class of dependence, the *scheduler-carried dependence*. We have described how this new dependence occurs and the necessary invariants required to avoid violating it, as well as implications of these invariants. Using these insights, we introduced Hive, an execution model and hardware architecture that implements a scalable priority queue with priority update operations. Hive achieves up to 2.8× speedup over Swarm at 256 cores, whereas software solutions fail to scale to such large system sizes. We envision this effort as one step toward our ultimate goal: designing software and hardware support for both speculative and non-speculative task-based programming and execution models.

References

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. ACM, 1247–1262. https://doi.org/10.1145/3373376.3378454

[2] Vitalii Aksenov, Dan Alistarh, and Janne H. Korhonen. 2020. Scalable Belief Propagation via Relaxed Scheduling. In *Proc. of the International Conference on Neural Information Processing Systems (NeurIPS)*. MIT Press, 22361–22372.

[3] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. 2018. Relaxed Schedulers Can Efficiently Parallelize Iterative Algorithms. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*. ACM, 377–386. https://doi.org/10.1145/3212734.3212756

[4] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. 2017. The Power of Choice in Priority Scheduling. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*. ACM, 283–292. https://doi.org/10.1145/3087801.3087810

[5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList:A scalable relaxed priority queue. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 11–20. https://doi.org/10.1145/2688500.2688523

[6] Dan Alistarh, Giorgi Nadiradze, and Nikita Koval. 2019. Efficiency Guarantees for Parallel Incremental Algorithms under Relaxed Schedulers. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 145–154. https://doi.org/10.1145/3323165.3323201

[7] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 177–189. https://doi.org/10.1145/567067.567085

[8] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large Scale Networks Fingerprinting and Visualization Using the K-Core Decomposition. In *Proc. of the International Conference on Neural Information Processing Systems (NeurIPS)*. MIT Press, 41–50.

[9] Analog Bits 2011. *4096 x 128 ternary CAM datasheet (28nm)*. Analog Bits. http://mail.analogbits.com/pdf/28nm_TCAM_Product_Brief.pdf

[10] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Computer Architecture and Compiler Optimizations (TACO)* 14, 2 (2017). https://doi.org/10.1145/3085572

[11] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 181–192. https://doi.org/10.1145/2145816.2145840

[12] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM.

[13] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface.

[14] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. OOPSLA-20*.

[15] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *Proc. of the International Symposium on Computer Architecture (ISCA-48)*. ACM/IEEE, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[16] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 293–304. https://doi.org/10.1145/3087556.3087580

[17] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. 2011. Dark Silicon and The End of Multicore Scaling. In *Proc. of the International Symposium on Computer Architecture (ISCA-38)*. ACM/IEEE, 122–134. https://doi.org/10.1109/MM.2012.17

[18] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J.* ACM 34, 3 (1987), 596–615. https://doi.org/10.1145/28869.28874

[19] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[20] Thomas Haigh, Mark Priestley, and Crispin Rope. 2014. Reconsidering the StoredProgram Concept. *IEEE Annals of the History of Computing* 36, 1 (2014), 4–17. https://doi.org/10.1109/MAHC.2013.56

[21] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM, 58–69. https://doi.org/10.1145/384265.291020

[22] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 3–12. https://doi.org/10.1145/2038037.1941557

[23] Muhammad Amber Hassaan, Donald Nguyen, and Keshav Pingali. 2015. Kinetic Dependence Graphs. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*. ACM, 457–471. https://doi.org/10.1145/2775054.2694363

[24] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-centric execution of speculative parallel programs. In *Proc. of the International Symposium on Microarchitecture (MICRO-49)*. IEEE/ACM, 1–13. https://doi.org/10.1109/MICRO.2016.7783708

[25] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. of the International Symposium on Microarchitecture (MICRO-48)*. IEEE/ACM, 228–241. https://doi.org/10.1145/2830772.2830777

[26] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2016. Unlocking ordered parallelism with the Swarm architecture. *IEEE Micro* 36, 3 (2016), 105–117. https://doi.org/10.1109/MM.2016.12

[27] Mark C. Jeffrey, Victor A. Ying, Suvinay Subramanian, Hyun Ryong Lee, Joel Emer, and Daniel Sanchez. 2018. Harmonizing speculative and non-speculative execution in architectures for ordered parallelism. In *Proc. of the International Symposium on Microarchitecture (MICRO-51)*. IEEE/ACM, 217–230. https://doi.org/10.1109/MICRO.2018.00026

[28] Venkata Krishnan and Josep Torrellas. 1999. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. Comput*. 48, 9 (1999), 866–880. https://doi.org/10.1109/12.795218

[29] Joseph B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc*. 7 (1956), 48–50. https://doi.org/10.2307/2033241

[30] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[31] Edward A. Lee. 2006. The Problem with Threads. *IEEE Computer* 39, 5 (2006).

[32] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority queues are not good concurrent priority

schedulers. In *Proc. of the European Conference on Parallel Processing (Euro-Par)*. Springer Berlin Heidelberg, 209–221. https://doi.org/10.1007/978-3-662-48096-0_17

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 190–200. https://doi.org/10.1145/1064978.1065034

[34] Fragkiskos D. Malliaros, Christos Giatsidis, Apostolos N. Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: theory, algorithms and applications. *The VLDB Journal* 29 (2020), 61–92. https://doi.org/10.1007/s00778-019-00587-4

[35] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *J.* ACM 30, 3 (1983), 417–427. https://doi.org/10.1145/2402.322385

[36] U. Meyer and P. Sanders. 2003. Delta-stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[37] Flaviano Morone, Kate Burleson-Lesser, H. A. Vinutha, Srikanth Sastry, and Hernán A. Makse. 2019. The jamming transition is a k-core percolation transition. *Physica A: Statistical Mechanics and its Applications* 516 (2019), 172–177. https://doi.org/10.1016/j.physa.2018.10.035

[38] Flaviano Morone, Gino Del Ferraro, and Hernán A. Makse. 2019. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics* 15 (2019), 95–102. https://doi.org/10.1038/s41567-018-0304-8

[39] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proc. of the Symposium on Operating Systems Principles (SOSP-24)*. ACM, 456–471. https://doi.org/10.1145/2517349.2522739

[40] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. 2005. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News* 33, 5 (2005), 45–50. https://doi.org/10.1145/1127577.1127587

[41] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 12–25. https://doi.org/10.1145/1993498.1993501

[42] Gilead Posluns. 2022. *A Speculative Hardware Scheduler Supporting Priority Updates*. Master's thesis. University of Toronto.

[43] Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. 2022. A Scalable Architecture for Reprioritizing Ordered Parallelism. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (ISCA '22). ACM/IEEE, New York, NY, USA, 437–453. https://doi.org/10.1145/3470496.3527387

[44] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. 2022. Multi-Queues Can Be State-of-the-Art Priority Schedulers. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 353–367. https://doi.org/10.1145/3503221.3508432

[45] R. C. Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

[46] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proc. of the International Conference on Supercomputing (ICS '05)*. ACM, 219–228. https://doi.org/10.1145/1088149.1088178

[47] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. 2005. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *Proc. of the*

International Conference on Supercomputing (ICS '05). ACM, 179–188. https://doi.org/10.1145/1088149.1088173

[48] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 80–82. https://doi.org/10.1145/2755573.2755616

[49] Kazuki Sakamoto and Tomohiko Furumoto. 2012. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 139–145.

[50] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287. https://doi.org/10.1016/0378-8733(83)90028-X

[51] Mohsin Shan and Omer Khan. 2021. Accelerating Concurrent Priority Scheduling Using Adaptive in-Hardware Task Distribution in Multicores. *IEEE Computer Architecture Letters* (CAL) 20, 1 (2021), 17–21. https://doi.org/10.1109/LCA.2020.3045670

[52] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core em dash Analysis — Patterns, Anomalies and Algorithms. In *Proc of the International Conference on Data Mining (ICDM)*. IEEE, 469–478. https://doi.org/ 10.1109/ICDM.2016.0058

[53] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement:The Problem Based Benchmark Suite. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 68–70. https://doi.org/10.1145/ 2312005.2312018

[54] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *Proc. of the International Symposium on Computer Architecture (ISCA-22)*. ACM/IEEE, 414–425. https://doi.org/10.1145/223982.224451

[55] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In *Proc. of the International Symposium on Computer Architecture (ISCA-27)*. ACM/IEEE, 1–12. https://doi.org/10.1145/339647.339650

[56] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An execution model for fine-grain nested speculative parallelism. In *Proc. of the International Symposium on Computer Architecture (ISCA-44)*. ACM/IEEE, 587–599. https://doi.org/10.1145/3079856.3080218

[57] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. 1999. The Superthreaded Processor Architecture. *IEEE Trans. Comput*. 48, 9 (1999), 881–902. https://doi.org/10.1109/12.795219

[58] Jean Vuillemin. 1978. A Data Structure for Manipulating Priority Queues. *Commun. ACM* 21, 4 (1978), 309–315. https://doi.org/10.1145/359460.359478

[59] Maurice V. Wilkes and William Renwick. 1949. The EDSAC - an Electronic Calculating Machine. *Journal of Scientific Instruments* 26, 12 (1949), 385–391. https://doi.org/10.1088/0950-7671/26/12/301

[60] J. W. J Williams. 1964. Algorithm 232 Heapsort. *Commun. ACM* 7, 6 (1964), 347–349. https://doi.org/10.1145/512274.512284

[61] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The Lock-Free k-LSM Relaxed Priority Queue. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 277–278. https://doi.org/10.1145/2688500.2688547

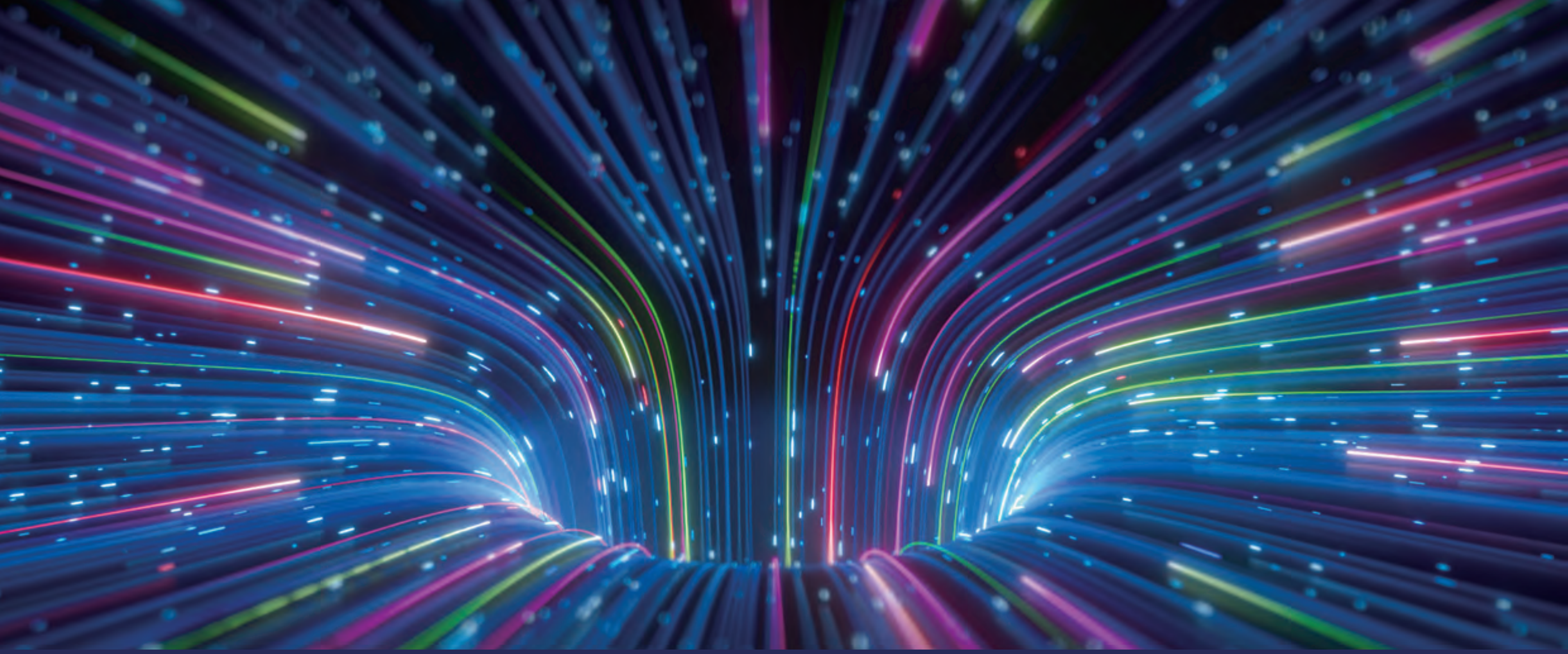
[62] Stefan Wuchty and Eivind Almaas. 2005. Peeling the yeast protein network. *Proteomics* 5 (2005), 444–449. Issue 2. https://doi.org/10.1002/pmic.200400962

[63] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2019. Understanding Priority-Based Scheduling of Graph Algorithms on a Shared-Memory Platform. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*. ACM, 1–14. https://doi.org/10.1145/3295500.3356160

[64] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. 2020. T4: Compiling sequential code for effective speculative parallelization in hardware. In *Proc. of the International Symposium on Computer Architecture (ISCA-47)*. ACM/IEEE, 159–172. https://doi.org/10.1109/ISCA45697.2020.00024

[65] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*. IEEE. https://doi.org/10.48550/ARXIV.1911.07260

[66] Tingzhe Zhou, Maged Michael, and Michael Spear. 2019. A Practical, Scalable, Relaxed Priority Queue. In *Proc. of the International Conference on Parallel Processing (ICPP)*. ACM, 1–10. https://doi.org/10.1145/3337821.3337911



Exploration of Hybrid Optical-Electrical Switching Networks in AI Training Clusters

Shengyu Shen, Di Chen, Wenkai Ling, Jingyan Wang, Tianchi Hu, Shanggang Xie, Tianxiang Chen, Yang Ren, Jifang He, Qihang Duan, Zeshan Chang, Weibin Lin, Xin Liu, Guangcan Mi

Research Dept, Computing Product Line

Abstract

Electrical switching networks in heavy computing scenarios such as AI and high-performance computing (HPC) are troubled by the high cost and power consumption of non-blocking fat trees. In addition, hash conflicts also cause severe performance loss. To resolve these issues, this paper proposes to build a large-scale, high-bandwidth, and flexible hybrid optical-electrical switching network by leveraging the large number of ports on the micro-electro-mechanical system (MEMS) optical cross-connect (OXC), the high-bandwidth feature of wavelength aggregation optical components, and the flexibility of a super node architecture. In addition, for the key AllReduce and AlltoAll communication operators, this paper proposes a new collective communication algorithm to adapt to the hybrid optical-electrical switching network. Experimental results and analysis show that in the ResNet50 data parallel training scenario, a 64-NPU prototype system provides the same performance as the optimal non-congestion electrical network. In addition, a cluster of 4K to 32K (K = 1024) NPUs reduces the end-to-end power consumption of the entire system by about 10%. To satisfy the requirements of future hybrid HPC/AI scenarios, we further propose the FAT-Dragonfly+ topology and pipelined collective communication algorithm based on existing work, pointing out a new direction for the future exploration of hybrid optical-electrical switching networks.

Keywords

AI, neural network, large model, distributed training, collective communication, optical communication, optical cross-connect

1 Introduction

In the AI field, by using simple neural-network architecture and combining big data sets, high computing power, and large-scale training parameters, a large neural network model surpasses complex algorithms in precision performance. In the natural language processing (NLP) field, the industry has embarked on a race for large models. The parameter number of large models increased from 340 million (in 2018) of Bidirectional Encoder Representations from Transformers (BERT) [1], 1.5 billion (in 2019) of Generative Pre-trained Transformer 2 (GPT-2) [2], and 175 billion (in 2020) of GPT-3 [3] to 1.6 trillion (in 2021) of Switch Transformer [4]. Huawei also released the Pangu-α model [5] in 2021, with up to 200 billion parameters.

The rapid growth of the model size poses higher requirements on hardware infrastructure such as computing power and memory. The supercomputer cluster jointly built by Microsoft and OpenAI has 285,000 central processing unit (CPU) cores and 10,000 graphics processing units (GPUs), providing a training platform[1] for AI models such as GPT-3. And Meta is building the AI Research SuperCluster (RSC) — a supercomputer for AI research — that has 760 computing nodes and 6080 NVIDIA DGX A100 GPUs. Meta plans to increase the number of GPUs in the RSC to 16,000[2] by 2022.

However, simply stacking machines does not necessarily increase the overall computing power of AI clusters. In the distributed training of a large model, a large training task is usually split into smaller parts and distributed to multiple nodes, with each node then distributing its part of the task to multiple neural processing units (NPUs). Because each NPU completes only a part of the task, they need to communicate with each other in order to exchange data and aggregate computation results. The high computing power of an AI cluster can be fully utilized only when computing and communication in the cluster are well coordinated.

Stacking large numbers of computing devices poses demanding challenges to NPUs interconnection network. To implement a high-bandwidth cluster network, existing AI clusters use high-speed network connection technologies. The network bandwidth of each GPU server in the supercomputer cluster built by Microsoft and OpenAI is 400 Gb/s. Each GPU in the RSC is connected to other GPUs through a 200 Gb/s HDR InfiniBand network. And DGX SuperPOD — a next-generation AI data center infrastructure platform released by NVIDIA — is equipped with a 400 Gb/s HDR InfiniBand network interface card (NIC) for each GPU.

However, as shown in Figure 1, there is a gap between the growth of computing power and that of bandwidth. In the future, distributed AI training models will have higher requirements on bandwidth. Each NPU will require a bandwidth of up to 10 Tb/s [6]. In fact, the trend of SerDes packet switching indicates that the SerDes rate improvement is limited by density and rate [7], which makes it difficult to achieve 10 Tbit/s bandwidth per port based on electrical switching chips. This limit is reflected in the following aspects:

1 Further improvement of the SerDes rate will significantly increase the transmission loss of electrical signals and increase the SerDes power consumption.

2 The cost of optical modules will become an important issue [8]. A large fraction of this cost is attributed to packaging and assembly, unlike that of electrical switching chips. And because the capacity of optical modules does not scale with the improvement of semiconductor process technologies, the cost of optical modules required to populate a data center switching chip now exceeds that of the switching chip itself.

3 The growth of bandwidth density has started to lose pace. To accommodate the required number of pluggable optical modules and the necessary thermal management after power consumption growth, the size of the switch enclosure has doubled [8].

On the other hand, substantial progress has been made on silicon photonic chips by integrating optical devices and electrical components into independent application-specific integrated circuit (ASIC) chips and using laser beams instead

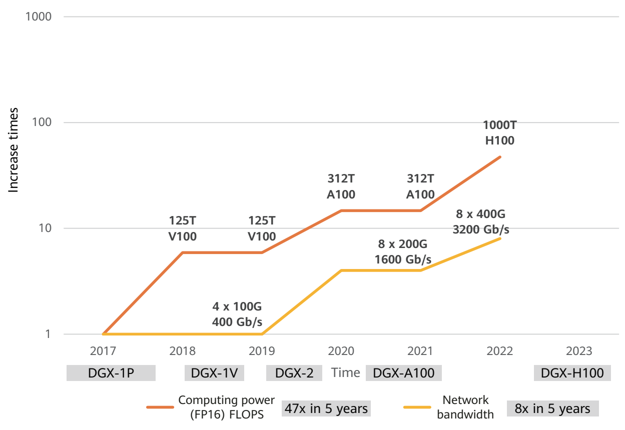


Figure 1 Growth gap between computing power and bandwidth

[1] Microsoft announces new supercomputer and lays out vision for future AI work: https://blogs.microsoft.com/ai/openai-azure-supercomputer/

[2] Meta works with NVIDIA to build massive AI research supercomputer: https://blogs.nvidia.com/blog/2022/01/24/meta-ai-supercomputer-dgx/

of electronic signals to improve bandwidth. Currently, the data transmission rate of silicon photonic chips is much higher than that of electrical signals, achieving an egress bandwidth in the Tb/s level [9]. This brings new opportunities for the hybrid optical-electrical networks of AI training clusters.

However, optical switching also faces some challenges. Currently, there are three mainstream optical cross-connect (OXC) technologies: micro-electro-mechanical system (MEMS) OXC, wavelength selective switch (WSS), and sub-µs fast optical cross-connect. The MEMS and WSS technologies, which take tens to hundreds of milliseconds to established new optical paths, can only be used in long-task scenarios and cannot meet the ns-/µs-level quick response requirements in AI training. In contrast, fast optical cross-connect can achieve ns-/µs-level switching time, but the port number cannot be further increased due to factors such as loss and packaging.

In addition, on a traditional electrical switching network, the equal-cost multi-path (ECMP) [10] algorithm randomly hashes traffic to multiple paths to balance traffic. However, this method does not consider the time and space distribution characteristics of the traffic of the application itself, leading to more than 50% bisection bandwidth loss caused by hash collisions [11]. This loss is unacceptable for bandwidth-hungry AI scenarios.

Based on the advantages of MEMS, fast optical cross-connect components, and electrical switching, we have designed a new architecture of heterogeneous hybrid optical-electrical switching networks. This architecture fully takes advantages of different optical and electrical technologies, supports a cluster size (number of NPUs/GPUs) of up to 128K, and achieves ns-/µs-level fast switching time. In addition, network traffic in AI scenarios is periodic and predictable elephant flows generated by clearly defined collective communication operators (such as AllReduce and AlltoAll). The communication mode is the same in each iteration of the training process. Compared with random and disordered traffic in traditional data centers, network traffic in AI scenarios greatly simplifies the communication algorithm and control plane design. Therefore, we propose to redesign the control plane for optical networks and adapted collective communication algorithms to optical networks, thereby avoiding the performance loss caused by hash collisions.

In the future, the hybrid optical-electrical switching networks of AI clusters will witness several development trends: ultra-high bandwidth, ultra-low latency, ultra-low power consumption, simplified configuration, and ultra-large scale. Our team will make useful explorations in these directions.

This paper is organized as follows: Section 1 analyzes the background and requirements of AI cluster hybrid optical-electrical switching networks. Section 2 introduces mainstream optical cross-connect technologies and focuses on the optical components suitable for AI training scenarios. Sections 3 and 4 describe the challenges faced by hybrid optical-electrical switching networks and the solutions to these challenges. Section 5 shows the prototype system and preliminary test results. Section 6 points out new exploration directions to further improve algorithm performance and adapt to a wider range of HPC scenarios with a hybrid optical-electrical switching network architecture. And finally, Section 7 summarizes related work.

2 Optical Communication Technologies

The core function of OXC is to implement optical channel connections from one or more input optical fiber ports to multiple output optical fiber ports. By using different switching control mechanisms, OXC can output an optical signal of one input optical fiber port from other optical fiber ports. That is, an optical signal is switched between output ports. Currently, OXC connection technologies are booming, and their major development directions are wavelength-level cross-connect and fiber-level port cross-connect. This section focuses on the MEMS OXC and sub-µs fast optical cross-connect technologies used in this solution.

2.1 MEMS OXC

The research on MEMS OXC in the industry started in the early 21st century. At the Optical Fiber Communication (OFC) Conference in 2001, Lucent announced the research on the prototype of MEMS optical switches with 1296 x 1296 ports [12]. From 2003 to 2004, Bell Labs and Lucent published the research work on MEMS optical switches with 256 x 256 ports [13, 14]. In 2012, Calient launched the MEMS OXC product S320[3] based on the electrostatic mechanism. The S320 provides 320 fiber port switching channels and applies to scenarios such as data center networks, software-defined networks (SDNs), and automatic testing. And then in 2016, Polatis (acquired by

[3] S320 product of Calient: https://www.calient.net/

Huber+Suhner) launched the MEMS OXC Series 7000[4] based on the piezoelectric mechanism. The Series 7000 supports the switching of 384 x 384 fiber ports.

In a MEMS OXC switching control mechanism, a MEMS structure is used to drive a reflector to change the propagation angle of light. This enables an optical signal to be arbitrarily switched to different output ports, as shown in Figure 2. The MEMS OXC consists of four core components: an input fiber collimator array, an input MEMS micromirror array, an output MEMS micromirror array, and an output fiber collimator array. The unit channels of the fiber collimator array are in one-to-one correlation with the unit channels of the MEMS micromirror array. The spatial optical path based on micromirror reflection mainly presents three characteristics of the MEMS OXC:

1 The number of ports is large. The MEMS reflector is processed using integrated circuit (IC) techniques to achieve a large-scale two-dimensional MEMS micromirror array, thereby increasing the number of MEMS OXC ports. In addition, a direct reflection optical structure determines that any input port of the MEMS OXC may establish a contentionless connection to any output port. Currently, MEMS OXC with a maximum of 384 inputs/outputs has been implemented in the industry.

2 The port capacity is large and is irrelevant to the rate, wavelength, and bandwidth of transmitted optical signals. The MEMS reflector changes the propagation direction of an optical signal by adjusting a spatial deflection angle, without processing the optical signal. Therefore, the propagated optical signals may have different wavelengths, rates, bandwidths, or modulation formats.

3 The switching speed of an optical switch is slow. A MEMS OXC controls beam deflection based on mechanical adjustment principles. Although the MEMS OXC has the advantages of low insertion loss and polarization independence, it requires millisecond-level time to switch. In addition, the reliability and stability of micromechanical structures also need to be improved.

In AI computing cluster scenarios, a MEMS OXC that features many ports, large capacity, and low power consumption can satisfy two key system requirements:

1 Large-scale networking with more than 400 ports should effectively support the interconnection between large-scale compute nodes, and should be flexibly reconstructed at the job and tenant levels.

[4] Series 7000 product of Polatis: https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabled.asp

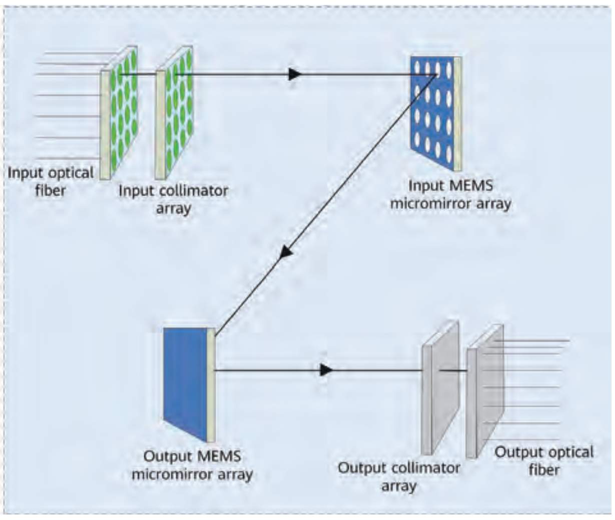


Figure 2 Working mechanism of a MEMS OXC

2 High bandwidth adaptability should cover different bandwidths and rates, and support further port bandwidth evolution to 100 Gb/s, 200 Gb/s, 400 Gb/s, and even 800 Gb/s.

Huawei has carried out research on MEMS OXC, covering MEMS micromirror array chips with many ports, high-density chip packaging, low-loss fiber collimators, multi-channel precision optical coupling, and micromirror control algorithms. It has also further explored engineering issues such as seismic reliability, highly efficient testing and calibration, and cost reduction of key components. Huawei has implemented a 400-port prototype that it has tested and verified in the AI cluster prototype system. To further meet the scale expansion requirements of an AI intelligent computing platform, Huawei is achieving breakthroughs in the 512 x 512–port MEMS OXC technology.

2.2 Sub-μs Fast Optical Cross-Connect

The fast cross-connect component using an optical switch supports point-to-point strict contentionless communication, and the optical switch has a switching speed at the ns to μs level, which can meet the requirements of fast switching in AI training. However, compared with a MEMS OXC, the fast cross-connect component has disadvantages in port number, component loss, packaging, and optical switch cost.

1 On-chip integrated optical switch

Based on which key technology is applied, on-chip integrated fast optical switches are classified into five types: thermo-optic effect, free carrier effect, Pockels effect, Kerr effect, and silicon-based MEMS (Si-MEMS).

The thermo-optic effect indicates that the refractive index of the material is regulated by using the temperature-sensitive characteristics of the lattice material structure. Silicon is an ideal thermo-optic effect carrier due to it having the highest temperature coefficient in common optical components. An optical switch made of silicon can achieve an ultra-compact size in the 100 µm level [15]. Manufacturing an overlay channel near the silicon waveguide enables heat diffusion to be further limited and a switch power consumption of 10 mW to be achieved. The thermo-optic switch can achieve a sub-µs switching latency by using an overshoot signal in an initial phase of heating and cooling and by placing a heat source as close as possible to a waveguide center [16]. The switching time of the 32 x 32–port on-chip integrated thermo-optic switch array reported in [15] is about 750 µs, and the loss is about 0.5 dB/cm.

The free carrier effect is also based on silicon materials. Because the band gap of silicon is relatively large, the intrinsic carrier concentration of pure silicon is relatively low and cannot be used as an effective means of regulating the refractive index. Therefore, boron and phosphorus ions are implanted into a silicon waveguide to provide carriers. Due to the diffusion-drift free movement of carriers, the material refractive index of carriers is generally regulated at the ns level. However, the injection of carriers increases the risk of material defects, narrows the band gap, and amplifies the material's absorption of light. As a result, the length of an optical switch with the carrier dispersion effect is at the 300 µm to mm level, with 0.5–1 dB extra loss and more than 3 dB degradation of port isolation. The switching latency of the 32 x 32 carrier optical switch reported in [17] is 1.2 ns, the port isolation is 18.1 dB, and the highest link loss is 16.5 dB.

Both the Pockels effect and Kerr effect are non-linear optical effects. When the electric field strength in the waveguide or the applied electric field strength is large enough, the material polarization phenomenon created by this field causes the refractive index of the material to change (electro-optic effect, which has been widely used in high-speed modulators [18]). The difference between the two effects is that the crystal structure with mirror symmetry and central symmetry has only the Kerr effect, whereas the non-central symmetry structure material has both the Pockels effect and Kerr effect. The electro-optic response time of an optical switch with a non-linear optical effect is at the ps to fs level, and no extra loss is generated. However, such an optical switch requires a higher drive voltage or a longer component length. The major challenge of a highly integrated electro-optic effect optical switch is how to balance a large number of ports and a low drive voltage while also ensuring low loss. While the switching latency of the SOI-based thin-film lithium niobate dual-polarized optical switch reported in [19] is about 300 ps, the 6 mm component length limits the integration scale and causes cost issues, and the switching voltage of about 10 V further increases the drive cost.

Si-MEMS has attracted more and more attention in recent years. It directly changes the physical spacing between waveguide objects based on the attraction/ rejection pattern of the suspended waveguide structure by electrostatic force, thereby changing the optical path. The switching speed of the Si-MEMS optical switch is at the sub-μs level. Compared with other technologies, Si-MEMS provides higher isolation and lower loss, and allows a more compact structure (supporting more than 64 x 64 ports). For example, the Si-MEMS optical switch reported in [20] has 240 x 240 ports. However, Si-MEMS relies on a mobile waveguide or metal electrode structure, which limits the reliability and durability of the optical switch. In addition, Si-MEMS requires multiple layers of waveguides to be coupled. Therefore, the preparation process is more complex and incompatible with the CMOS process.

2 Fast optical cross-connect based on optical switches

As mentioned earlier, the on-chip integrated optical switch supports strict contentionless switching with a ns-level latency. It also provides 1 x N and N x N optical cross-connect specifications. Common N x N fast optical cross-connect specifications based on optical switches include N x N strict contentionless optical switch cross-connect and N x N aggregation optical switch cross-connect.

N x N strict contentionless optical switch cross-connect: As shown in Figure 3, nodes are interconnected by using an N x N optical switch, and one-to-one communication between nodes is implemented at any time by controlling switching of the optical switch.

N x N aggregation optical switch cross-connect: As shown in Figure 4, the N x N aggregation optical switch cross-connect works with multiple fixed-wavelength lasers to support one-to-one and many-to-one switching between N inputs and N outputs. That is, it supports both optical switching and bandwidth aggregation.

The fast optical cross-connect based on optical switches uses the port switching function of optical switches to implement the switching and aggregation functions. And because other components (such as the light source and modulator) and the receive end of the system do not need to be adjusted, no additional latency is introduced. Therefore, the switching latency of the optical switch (i.e., the end-to-end physical switching latency) can reach the ns level. Nonetheless, the on-chip integrated optical switch technology needs to be further explored to resolve issues such as port number, component loss, packaging, and costs.

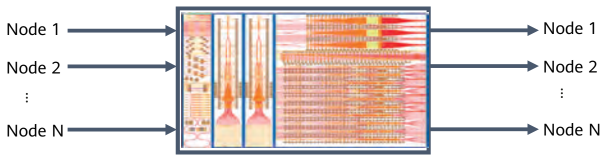


Figure 3 N x N optical switch cross-connect, one-to-one input/output

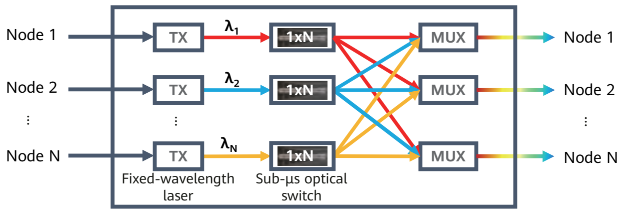


Figure 4 N x N aggregation optical switch cross-connect, many-to-one input/output

2.3 Optical Cross-Connect Technology Applicable to AI Scenarios

Mainstream optical cross-connect technologies today include wavelength-level cross-connect and fiber-level port cross-connect. Port cross-connect technologies can be classified based on the use of either spatial light or waveguide light. There are significant differences between optical cross-connect technologies in terms of optical switch switching latency, port scale, and maturity, as listed in Table 1.

In AI large model training scenarios, optical cross-connection configurations are divided into two phases. The first phase is **pre-configuration**. In this phase, the optical switch is pre-configured upon task submission. After the pre-configuration is completed, there is no need to modify optical cross-connection configurations throughout the entire training process. The MEMS OXC technology with a large number of ports, a large capacity, and a millisecond-level configuration latency can be used. The second phase is **on-the-fly reconfiguration**. While a task is running, optical cross-connections are configured on-the-fly to adjust the destination of traffic. In this phase, the fast optical cross-connect technology with ns-/μs-level configuration latency is required.

Table 1 Mainstream optical cross-connect technologies

Category	Type	Switching Time	Port Scale	Risk
Wavelength switching	**Fast tunable laser (thermal tuning)**	ms to s	Limited by the number of wavelengths	None
	Fast tunable laser (electrical tuning)	ns	Limited by the number of wavelengths	Wavelength locking issue: Wavelength drift may occur. In this case, wavelength locking needs to be performed in closed-loop tuning mode, which affects the switching time.
	SOA wavelength selection	ns	Limited by the number of wavelengths	Restricted by the multi-wavelength light source process and SOA array, the scale is limited.
	WSS	100 ms	Limited by the number of wavelengths	None
	Microring wavelength routing (thermal tuning)	ms	Limited by the number of wavelengths	Trade-off is required for the microring filtering feature and isolation, which may affect the scale.
	Microring wavelength routing (electrical tuning)	ns	Limited by the number of wavelengths	(1) Trade-off is required for the microring filtering feature and isolation, which may affect the scale. (2) The microring may drift and needs to be tuned in a closed-loop manner, which affects the switching time.
Port switching (spatial light)	**MEMS OXC**	10 ms	320/512	None
	Piezoelectric OXC	ms to 10 ms	384	High cost
	2D mechanical switch	ms to s	2 x 2	Relatively high cost and large size
	2D MEMS switch	ms	8/16	None
	Magneto optical switch	30 μs to 50 μs	4/8	High cost
	High-voltage lithium niobate switch	10 ns to 300 ns	8 x 12	High cost, large size, and requiring 300 V drive voltage
	Low-voltage lithium niobate switch	10 ns	2 x 2	High cost and large size
Port switching (waveguide light)	**Silicon photonic MEMS**	700 ns	64/128	The chip manufacturing process requires silicon photonics and MEMS techniques, which may require iterative optimization. A drive voltage of 40 V is required.
	220 nm silicon photonics	10 ns	16/32	(1) Temperature drift or aging drift may occur. Closed-loop tuning is required, which affects the switching time. (2) The isolation is about –17 dB, and the architecture needs to be optimized.
	3 μm silicon photonics	100 ns to 300 ns	8/16	
	6 μm silicon photonics	400 ns	2	
	Thin-film lithium niobate (TFLN)	ns	8/16	The wafer cost is high.
	Electro-optic polymer	ns	16/32	Aging issues may occur.
	Optical splitter + SOA	< 1 ns	1 x 8	High ASE noise is introduced, affecting the signal-to-noise ratio (SNR).

The combination of the preceding two components, new application algorithms, and system control can form an efficient hybrid optical-electrical switching network architecture for AI training clusters. This heterogeneous optical cross-connect networking architecture satisfies the requirements of large-scale network and supports ns-/μs-level switching time. It has high flexibility and maximizes the advantages and potential of optical cross-connect.

3 Challenges Faced by AI Optical Training

It is not easy to train large models, especially those with hundreds of billions or even trillions of parameters. For example, the Pangu-α model has 200 billion parameters, consuming 750 GB of memory just for the model parameters. A large amount of additional memory space is also required for information such as datasets, gradients, and optimizer status during training [5]. On the other hand, NVIDIA's most advanced H100 GPU has a memory capacity of only 80 GB. Limited by the memory wall and overall power consumption of NPUs, the growth rate of an NPU's memory capacity cannot match the growth rate of the scale of large models. Furthermore, the computing power of a single NPU cannot meet the massive computing requirements of large models. Therefore, using the high computing power and large memory of multi-node AI clusters to accelerate large model training has become a mainstream choice, and distributed training is imperative.

3.1 Communication in Distributed Training

Common distributed training modes include data parallelism, operator-level model parallelism, pipeline model parallelism, and mixture of expert (MoE) parallelism.

Data parallelism: In this mode, the distributed training splits the dataset into smaller parts and sends these parts to each training group. Each training group consists of one or more NPUs and has a complete model. The AllReduce operation is performed between training groups to aggregate gradients.

Operator-level model parallelism: In this mode, the distributed training splits a model and sends a complete dataset to each training group. Each training group consists of one or more NPUs, but has only a part of the complete model.

Pipeline model parallelism: Unlike operator-level model parallelism, this mode splits a complete model by layer and places each complete layer into different training groups. Each training group consists of one or more NPUs and one or more complete model layers, but has only a part of the complete model.

MoE parallelism: This mode selects different parameters for different samples (unlike the traditional neural network model, which uses the same parameters for all sample data). Therefore, only some parameters are trained for each sample, but the model parameter number is greatly increased. In MoE parallelism, because the feature vectors of a node are sent to experts of other nodes through the routing algorithm, AlltoAll traffic is introduced.

The parallelism policy determines the communication between NPUs, and this communication determines the scheduling between links. In AI training, collective communication is generally used to implement data exchange and result aggregation between NPUs. Common collective communication operations include AlltoAll, AllReduce, ReduceScatter, Broadcast, and AllGather. In distributed training, these collective communication operations generally use hierarchical algorithms to implement intra-node and inter-node communication. This section uses AllReduce as an example to describe the implementation of hierarchical collective communication algorithms on existing electrical switching networks.

Classical AllReduce algorithms can be classified into two types: One type is ring-based, which is usually used in scenarios where the data volume is large (for example, larger than 1 MB) or the number of NPUs is relative small (for example, smaller than 16). The other type is tree-based, such as the recursive halving and doubling (HD AllReduce) algorithm, which is more suitable for scenarios where the data volume is small or the number of NPUs is relative large. Because intra-node bandwidth is much higher than inter-node bandwidth, a hierarchical AllReduce algorithm is generally used in distributed applications to reduce the number of inter-node communication times and communication overheads.

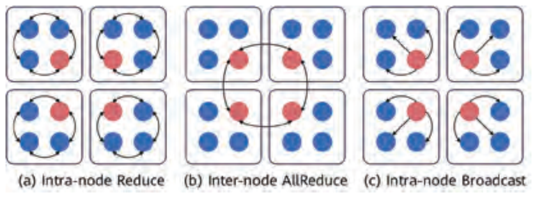


Figure 5 Hierarchical ring AllReduce algorithm proposed by Tencent

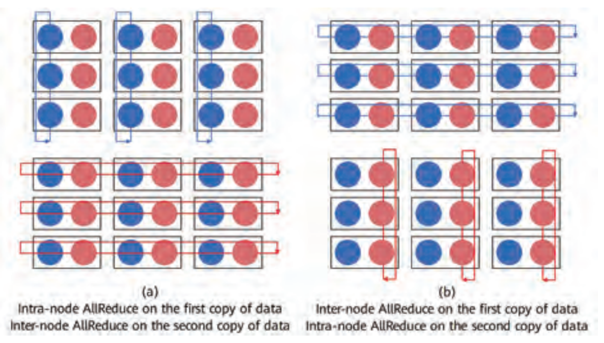

(a)
Intra-node AllReduce on the first copy of data
Inter-node AllReduce on the second copy of data
(b)
Inter-node AllReduce on the first copy of data
Intra-node AllReduce on the second copy of data

Figure 6 2D-mesh hierarchical AllReduce algorithm proposed by Google

A hierarchical algorithm usually uses high bandwidth within a node to transmit as much data as possible in the node. For example, Figure 5 shows a hierarchical ring AllReduce algorithm proposed by Tencent [21]. As shown in the figure, there are four nodes, and each node has four NPUs. The main steps involved in this algorithm are as follows:

1 Intra-node Reduce.

2 Inter-node AllReduce.

3 Intra-node Broadcast.

Figure 6 shows the 2D-mesh hierarchical AllReduce algorithm proposed by Google [22]. As shown in the figure, there are three nodes, each node has three NPU training cards, and data in each training card is divided into two copies. The main steps involved in this algorithm are as follows:

1 Intra-node AllReduce on the first copy of data and inter-node AllReduce on the second copy of data.

2 Inter-node AllReduce on the first copy of data and intra-node AllReduce on the second copy of data.

In each step, the two AllReduce operations are performed concurrently and do not interfere with each other.

3.2 Interconnection Issues Faced by AI Optical Training

Optical buffering is difficult to implement. Consequently, an optical switch usually implements circuit switching, whereby NPUs are connected in a point-to-point manner. In an optical network with circuit switching, only one NPU can be connected to another NPU at any one time. The limited number of switch ports means that the number of connections between NPUs is also limited, and therefore full interconnection between NPUs cannot be implemented. In addition, communication traffic in the AI large model includes not only simple point-to-point communication, but also complex collective communication. Adapting an existing collective communication algorithm to an optical network architecture has become an issue that urgently needs to be resolved.

Taking the Pangu-α model as an example, Figure 7 shows the schematic diagram of running the Pangu-α model in pipeline parallelism mode. There are 8 servers in the training cluster, and each server is configured with 8 NPUs (64 NPUs in total, numbered from 0 to 63). In distributed training, two channels of parallel data are used — the NPUs in blue are applied to one channel, and those in yellow are applied to the other channel. Each channel has a complete model and runs on multiple servers in pipeline parallelism mode, and each server runs a stage in the pipeline. Eight NPUs in each stage are used for operator-level model parallelism.

Table 2 shows the communication traffic in the Pangu-α model in pipeline parallelism mode. In the training process, there are a total of five communicators:

1 Four inter-node NPUs.

2 Eight intra-node NPUs.

3 16 inter-node NPUs.

Table 2 Pipeline parallelism communicators

Communicator	Intra-communicator NPU No. (Using the NPU Where Rank 16 Resides as an Example)	Communication Type
Four inter-node NPUs (pipeline communicator)	0-16-32-48	Send/Recv
Eight intra-node NPUs (operator-level model parallelism communicator)	16-17-18-19-20-21-22-23	AllReduce
		AllGather
		ReduceScatter
16 inter-node NPUs (data parallelism communicator)	16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-31	AllGather
		AllReduce
Two inter-node NPUs (data parallelism communicator)	16-24	AllReduce
		AllGather
		ReduceScatter
64 global NPUs	0-1-2-...-63	AllReduce

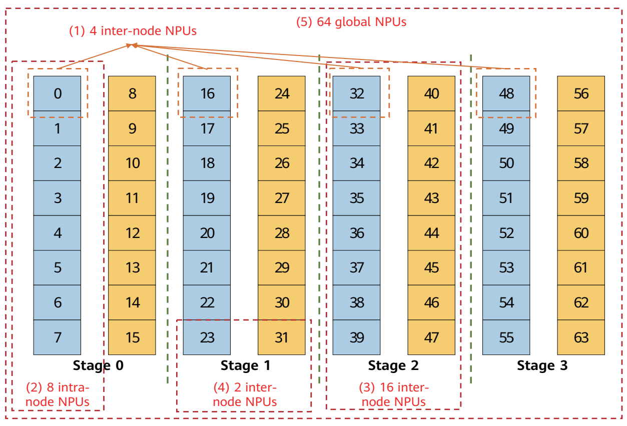


Figure 7 Pangu-α pipeline parallelism training cluster

4 Two inter-node NPUs.

5 64 global NPUs.

Taking NPUs 16 as an example, the communication types generated by them in different communicators diverse greatly. In an optical cross-connect network, connecting every communication link for complex and diversified communication traffic types has become a fundamental challenge.

4 AI Optical Training Solution

We propose a hybrid optical-electrical switching network architecture for next-generation machine learning cluster systems. Combining the advantages of a large number of MEMS ports and a low switching latency of sub-μs fast optical cross-connect, this heterogeneous hybrid optical-electrical switching network architecture can expand the cluster scale (number of NPUs/GPUs) to 128K (K = 1024) and meet requirements for ns-/μs-level fast link switching. Sub-μs fast optical cross-connect components are used to implement fast optical link switching within a cluster node, while MEMS components are used to expand the cluster scale between cluster nodes.

In addition, traffic in AI applications is regular, periodic, and predictable, and links between nodes do not need to be frequently switched. This presents a unique opportunity for the efficient use of optical components, offering a way to greatly simplify the complex plane control logic of traditional data centers. A parallel training policy determines link scheduling, and the latter determines the operation logic of the control plane. Therefore, for our proposed architecture, we have redesigned the operation logic of the control plane and adapted collective communication algorithms to optical topology networking on the premise of being compatible with the existing parallel training policy, thereby building an efficient end-to-end (E2E) optical training path.

4.1 AI Optical Training Architecture

Figure 8 shows the hybrid optical-electrical switching network architecture of an AI cluster. The entire cluster can be divided into three layers.

1 The first layer is intra-node. Intra-node interconnection is implemented through high-bandwidth interconnection protocols (such as Nvlink[5], CXL[6], and UB [23]), and all memory within the node can be addressed in a unified manner. Intra-node NPUs are managed by the same operating system (OS). NPUs in this layer are interconnected through electrical switching, and the cable transmission distance is within 1 m, achieving ultra-high bandwidth between intra-node NPUs.

2 The second layer is within a cabinet or a Performance-Optimized Datacenter (POD), which is a super node consisting of 32 nodes. NPUs in this layer are interconnected using high-bandwidth interconnection protocols (such as Nvlink, CXL, and UB), and their memory is also addressed in a unified manner. Each node in the cabinet is managed by an independent OS. NPUs at this layer are interconnected through electrical switching, and the cable transmission distance is within 10 m. The first and second layers form a non-blocking fat tree structure.

3 The third layer is inter-cabinet. Cabinets are directly connected by long-distance optical fibers through the network ports on NPUs. Each network port in a cabinet has a single unique port index. Network ports with the same index in different cabinets are connected to the same MEMS OXC. After the OXC is configured, these network ports form a full-mesh network. One NPU can provide two network ports, and one super node can provide 512 network ports. In this way, 512 super nodes can be interconnected to build an AI cluster with 128K NPUs. At the third layer, interconnection is implemented through a high-bandwidth remote direct memory access (RDMA) network (such as InfiniBand[7] and RoCEv2[8]). In addition, to cope with the bursty traffic between cabinets, the wavelength fast optical cross-connect devices mentioned in Section 2.2 are

[5] Nvlink: https://en.wikipedia.org/wiki/NVLink
[6] CXL: https://en.wikipedia.org/wiki/Compute_Express_Link
[7] InfiniBand: https://en.wikipedia.org/wiki/InfiniBand
[8] RoCEv2: https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet

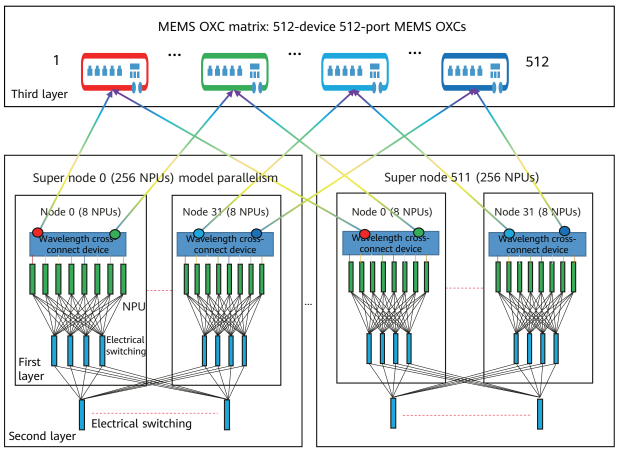

MEMS OXC matrix: 512-device 512-port MEMS OXCs
1
512
Third layer
Super node 0 (256 NPUs) model parallelism
Super node 511 (256 NPUs)
Node 0 (8 NPUs)
Node 31 (8 NPUs)
Wavelength cross-connect device
NPU
Electrical switching
First layer
Second layer
Electrical switching

Figure 8 Architecture of AI optical training networking

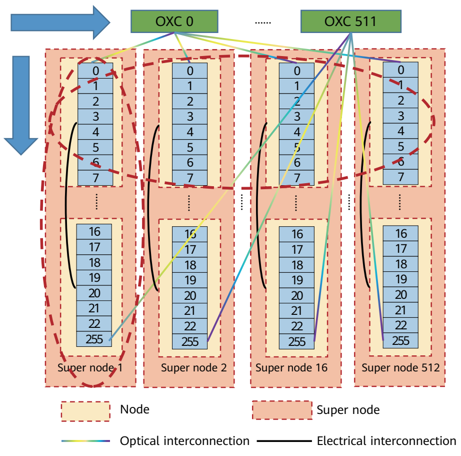


Figure 9 Dual-plane communication

used between cabinets, ensuring high-bandwidth traffic aggregation between them. Real-time adjustment of the wavelength cross-connect devices during the running process enables the bandwidth of the network port to be aggregated. The specific control process is described in the next section.

The aforementioned three-layer structure form an AI training cluster architecture. The network structure of the entire cluster can be logically divided into two communication planes, as shown in Figure 9. The first plane is the electrical plane formed by the first and second layers. Similar to NVIDIA DGX-H100 SuperPOD[9], each super node on this plane has up to 256 NPUs. The second plane is the all-optical interconnection plane formed by network ports on NPUs, with up to 512 super nodes on this plane. The two planes run independently and do not interfere with each other. Moreover, the wavelength cross-connect devices of the all-optical interconnection plane use the technologies and components designed for co-packaged optics/near-package optics (CPO/NPO). This makes it possible to meet the requirements of direct light output on NPUs in the future and share the industry chain to reduce costs.

Compared with the traditional electrical network architecture, this AI cluster network architecture has the following advantages:

1 Optical interconnection is used to build hard pipes. Wavelength cross-connect devices are used to prevent traffic conflicts between eight NPUs across super nodes.

2 The full-mesh networking of this architecture reduces the cost of optical modules by half.

3 MEMS OXC port switching avoids the need to design large-capacity electrical switches that require heavy investment but are seldom sold, thereby reducing non-recurring engineering (NRE) costs and power consumption.

4 The heterogeneous optical cross-connect architecture can match the evolution of high bandwidth (200 Gb/s or higher per lane) in the future on relative long distances.

[9] NVIDIA DGX SuperPOD: https://www.nvidia.com/en-us/data-center/dgx-superpod/

4.2 Control Plane of a MEMS OXC

A MEMS OXC features many ports, large capacity, and low power consumption. It is mainly used to connect nodes or super nodes in order to expand the scale of AI clusters. The main function of the MEMS OXC control plane is to establish optical paths for multiple NPUs/nodes of a tenant or task in the pre-configuration phase. After the pre-configuration is completed, there is no need to modify the connection relationship between nodes throughout the entire training process, and the requirement on the configuration latency of the MEMS OXC is relatively flexible.

4.2.1 Optical Management and Control Based on SDN

In a multi-tenant and multi-task environment, an AI cluster is divided into multiple independent small clusters for different tenants. In the tenant or task assignment phase, the MEMS OXC needs to be configured using an SDN controller to connect physical optical paths for NPUs of a tenant or task. It is necessary to ensure that there is at least an optical path between any two nodes. After the optical path is configured, the SDN controller transmits topology connection information to each compute node.

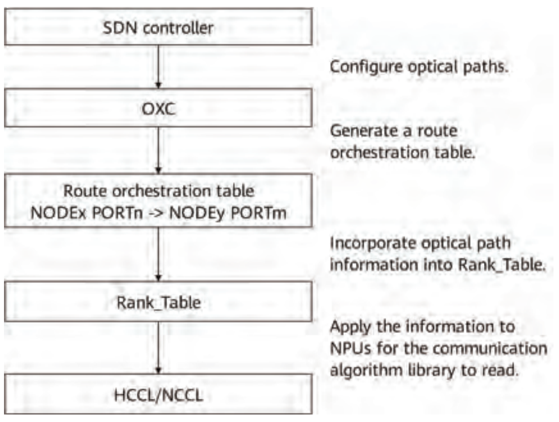

SDN controller
OXC
Route orchestration table
NODEx PORTn -> NODEy PORTm
Rank_Table
HCCL/NCCL
Configure optical paths.
Generate a route orchestration table.
Incorporate optical path information into Rank_Table.
Apply the information to NPUs for the communication algorithm library to read.

Figure 10 Topology information transmission

In this proposal, a route orchestration table is embedded in the cluster scheduling software (such as ModelArts[10]). The configured optical path information is incorporated into the Rank_Table of each NPU through the route orchestration table and applied to the corresponding NPU before communication starts. Figure 10 shows the process of transmitting topology information. In a training process, a communication operator in the corresponding communication library (such as NCCL or HCCL) switches an optical switch according to the route orchestration table, and dynamically adjusts the traffic destination.

Figure 11 shows how a tenant establishes an optical link based on the route orchestration table. Different tenants select NPUs among multiple nodes. Note that the red line between each two NPUs represents a multi-wavelength link that passes through the fast optical switches controlled by the two NPUs (it does not represent the data flow passing through the two NPUs). This means that each such multi-wavelength link can carry *N* data flows (*N* ranges from 8 to 16 based on the current component features). During the configuration, only the connections of the MEMS OXCs on an idle plane need to be configured. A tenant does not need to configure a connection between two nodes if the connection has been configured by another tenant. After the configuration is complete, the egress of the node where the NPU resides and the peer node ID corresponding to the egress need to be configured for the RANK_TABLE of the task or tenant, as shown in Table 3. This node's route orchestration table needs to be configured for the RANK_TABLE of each NPU so that the switching path of the optical switch can be selected based on the route orchestration table during communication (described in Section 4.2.1).

[10] ModelArts: https://www.huaweicloud.com/intl/en-us/product/modelarts.html

4.2.2 Full-Mesh Networking Configuration Policy

For the general configuration of large-scale MEMS OXCs on the entire network, this proposal provides a full-mesh connection configuration policy. This policy can quickly calculate the optical paths between nodes in multiple scenarios (with different numbers of nodes and NPUs in nodes).

In the case of an odd number of nodes, the current collective communication algorithm is a plane-based one. Therefore, in a single plane (e.g., a plane formed by all NPUs with the same index numbered in nodes), if every two NPUs are connected, there will be one NPU that cannot be connected to any other NPU. That is, the number of

Table 3 Node 0 route orchestration table

Node 0 Route Orchestration Table	
Destination Node ID	**Optical Switch ID**
Node 2	DEMUX 1/MUX 1
Node 3	DEMUX 4/MUX 4
Node 1	DEMUX 5/MUX 5

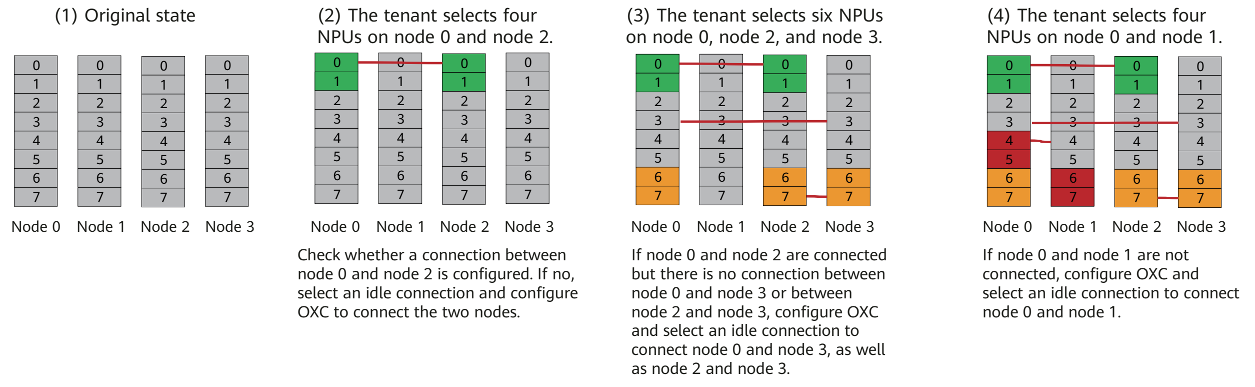

(1) Original state
(2) The tenant selects four NPUs on node 0 and node 2.
(3) The tenant selects six NPUs on node 0, node 2, and node 3.
(4) The tenant selects four NPUs on node 0 and node 1.
0
1
2
3
4
5
6
7
Node 0
Node 1
Node 2
Node 3
Check whether a connection between node 0 and node 2 is configured. If no, select an idle connection and configure OXC to connect the two nodes.
If node 0 and node 2 are connected but there is no connection between node 0 and node 3 or between node 2 and node 3, configure OXC and select an idle connection to connect node 0 and node 3, as well as node 2 and node 3.
If node 0 and node 1 are not connected, configure OXC and select an idle connection to connect node 0 and node 1.

Figure 11 Tenant configuration process

connections on a single plane decreases to *(N – 1)/2*. In order to implement full-mesh interconnection between nodes on multiple planes, *N* cards are required in each node. For example, if there are 7 nodes, 21 connections are required theoretically to achieve interconnection between every two nodes. Because a single plane can provide only three connections, seven NPUs are required in each node to implement full-mesh interconnection between nodes.

In the case of an even number of nodes, each plane can provide *N/2* connections. Therefore, only *N – 1* NPUs are required in each node to implement full-mesh interconnection between nodes. For example, if there are eight nodes, seven NPUs are required in each node to implement full-mesh interconnection between nodes on multiple planes. The total number of connections is 28.

The number of NPUs required in a node is denoted as N_{dev}, and the number of nodes is denoted as N_{node}. dev_{src} and dev_{dst} indicate the source NPU ID and destination NPU ID of the two interconnected NPUs respectively, with dev_{src}, $dev_{dst} \in [0, N_{dev})$. $node_{src}$ and $node_{dst}$ indicate the source NPU node ID and destination NPU node ID of the two interconnected NPUs respectively, with $node_{src}$, $node_{dst} \in [0, N_{node})$. The network-wide interconnection problem under the optical architecture can be formulated as, assuming that the source NPU node ID and source NPU ID are known, how to obtain the destination NPU node ID and destination NPU ID.

A hierarchical collective communication policy usually adopts the plane-based algorithm between nodes. The NPUs with the same number on all nodes form a plane. It can be seen that the number of planes in the system is equal to the number of NPUs in each node. Therefore, during optical interconnection configuration, only NPUs on the same plane need to be connected between nodes. That is, the source NPU (numbered dev_{src} in the node) and destination NPU (numbered dev_{dst} in the node) have the same number in the node, as shown in Formula (1).

$$dev_{dst} \equiv dev_{src} \tag{1}$$

As described above, the number of NPUs required to implement interconnection differs depending on whether the number of nodes is even or odd. When the number of nodes (*N*) is even, only *N – 1* NPUs are required in a node to implement full interconnection between nodes. When the number of nodes (*N*) is odd, at least *N* NPUs in a node are required to implement full interconnection between nodes. Therefore, for the scenarios with different numbers of nodes, different connection configuration policies can be used, as listed in Table 4.

Once these formulas are normalized, the configuration policy for an odd number of nodes can also be applied to the case of an even number of nodes, but in this policy the number of NPUs required in a node is not the optimal one. During the optical path configuration of a specific networking topology, configuration policies listed in Table 4 can be flexibly selected.

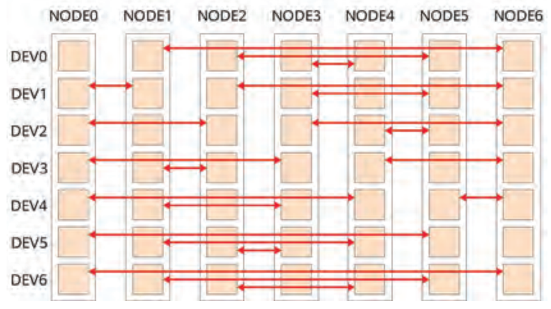


Figure 12 All-optical interconnection algorithm connections when there are 7 nodes and 7 NPUs in each node

Table 4 Inter-node interconnection configuration policies for different numbers of nodes

Number of Nodes	Condition (Conditions are matched from top to bottom.)	Destination Node ID
Even number	$node_{src} \equiv N_{dev}$ and $\frac{dev_{src}}{2} \equiv \left\lceil \frac{dev_{src}}{2} \right\rceil$	$node_{dst} = \frac{dev_{src}+N_{node}}{2}$
	$node_{src} \equiv N_{dev}$ and $\frac{dev_{src}}{2} \neq \left\lceil \frac{dev_{src}}{2} \right\rceil$	$node_{dst} = \frac{dev_{src}+1}{2}$
	$(dev_{src} + 1 - node_{src})\ mod\ N_{dev} \equiv node_{src}$	$node_{dst} = N_{dev}$
	None of the above conditions is met.	$node_{dst} = (dev_{src} + 1 - node_{src})\ mod\ N_{dev}$
Odd number	$(dev_{src} - node_{src})\ mod\ N_{node} \neq node_{src}$	$node_{dst} = (dev_{src} - node_{src})\ mod\ N_{node}$
	$(dev_{src} - node_{src})\ mod\ N_{node} \equiv node_{src}$	No destination node. That is, this NPU does not participate in inter-node interconnection or is user-defined.

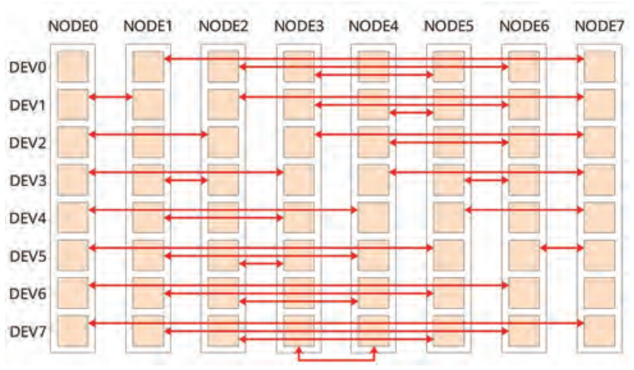


Figure 13 All-optical interconnection algorithm connections when there are 8 nodes and 8 NPUs in each node

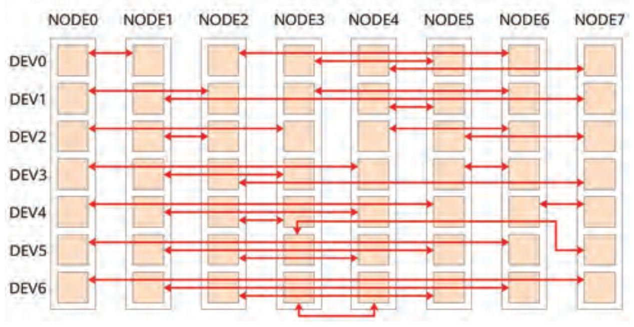


Figure 14 All-optical interconnection algorithm connections when there are 8 nodes and 7 NPUs in each node

According to the preceding formulas, when both the number of nodes and the number of NPUs in each node are odd numbers (e.g., there are 7 nodes and 7 NPUs in each node), the schematic diagram of connections for all-optical interconnection is shown in Figure 12.

When both the number of nodes and number of NPUs in each node are even numbers (e.g., there are 8 nodes and 8 NPUs in each node), the schematic diagram of connections for all-optical interconnection is shown in Figure 13.

When the number of nodes is an even number and the number of NPUs in each node is an odd number (e.g., there are 8 nodes and 7 NPUs in each node), the schematic diagram of connections for all-optical interconnection is shown in Figure 14.

4.3 Control Plane of Fast Optical Cross-Connect Components

Fast optical cross-connect components support point-to-point (P2P) strict non-blocking communication, and can implement ns-/μs-level optical switch switching. They are mainly used to quickly adjust traffic destinations in AI training. For fast optical cross-connect components, the main function of the control plane is to dynamically switch a connection between NPUs in cooperation with a collective communication algorithm of optical interconnection in a runtime phase. In complex large models and HPC applications, the type and size of traffic change dramatically, and links are frequently re-established between NPUs. These characteristics pose high requirements on the control latency of fast optical cross-connect components.

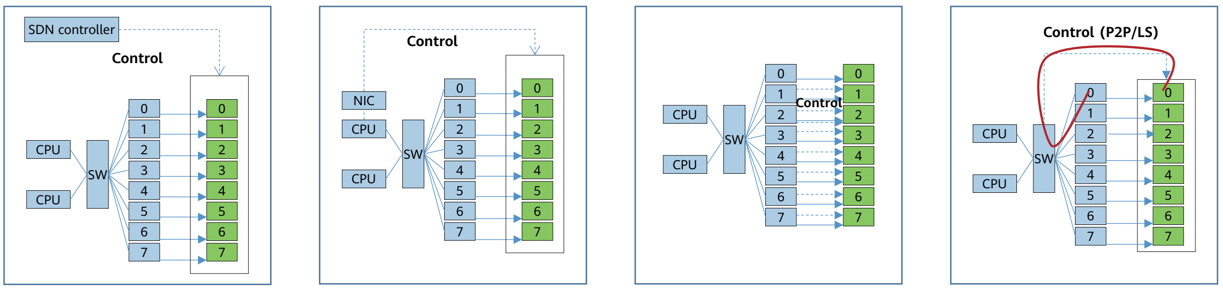
4.3.1 Optical Switch Device Forms

In a cluster network, an optical cross-connect device is generally used as a network device. A network device is usually controlled by an SDN controller, as shown in Figure 15a, and the control time is generally dozens of milliseconds or even seconds. The SDN controller can only perform static adjustment during initialization, but cannot perform real-time adjustment on a specific data flow in a training process. Therefore, the control mode based on an SDN controller can be used only for tenant-level configuration, but cannot be used for fast optical cross-connect control.

AI training uses NPUs as compute units. The CPU reads samples from memory and transmits the samples to an NPU for computing. The entire computing process is completed within the NPU. Parameters are automatically transmitted between NPUs without CPU intervention. If the fast optical cross-connect component is controlled by a CPU-based controller, as shown in Figure 15b, the NPU will frequently communicate with the CPU in a training process. The CPU-based control policy causes the end-to-end latency overhead to reach dozens of microseconds, which cannot meet the dynamic adjustment requirement in the training process.

In order to achieve microsecond-level end-to-end optical switch switching latency, we have considered the proposal in Figure 15c and Figure 15d. Figure 15c uses a network interface card (NIC) within an NPU for in-band control, and an in-band control command is sent to the corresponding optical cross-connect switch by the NPU through its NIC. However, in this proposal, the optical cross-connect switch board needs the same number of control interfaces as that of NPUs, and the controller implementation is relatively complex. In addition, the NPU cannot send in-band control commands to other optical switches or receive traffic from any NPU.

Based on these considerations, Figure 15d uses an optical switch board as an out-of-band controller. In this proposal,



(a) SDN controller (b) CPU-based controller (c) In-band controller (d) Out-of-band controller

Figure 15 Optical switch device forms

Table 5 Comparison of optical switch device forms

Controller Type	Application Scenario	Advantage	Disadvantage
Unified SDN controller (out-of-band SDN controller)	Application level Tenant level	(1) Global control. (2) The optical switch board requires only one control interface, which is easy to implement.	(1) Only the administrator has the required permission, and applications cannot be adjusted. (2) The latency overhead of the control operation is high.
Node CPU–based controller	Application level Tenant level	The optical switch board requires only one control interface, which is easy to implement.	(1) Security is poor. When NPUs are randomly selected for each tenant or task, different tenants may control other optical switches. (2) In offload mode, the NPU needs to interact with the CPU to support flow-level switching, resulting in high latency overhead.
Intelligent NIC controller (in-band controller)	Flow-level control Isolation between NPUs	The control is performed on the data plane, and the time overhead is low.	(1) The optical switch board has too many control interfaces. (2) The NPU needs to provide additional control interfaces.
Optical switch board controller (in-band controller)	Flow-level control Isolation between NPUs	(1) The control is performed on the data plane, and the latency overhead is low. (2) The optical switch board requires only one control interface, which is easy to implement.	The optical switch board must be presented as an endpoint or a peer device to the NPU.

the optical switch board is presented as a device of the node where the NPU resides, and it is tightly coupled with the NPU. The NPU and optical switch board can access each other in P2P mode. Before inter-node NPUs communicate with each other, they send instructions to control the switching of the optical switch. The NPUs can communicate with each other only after the switching is complete. Out-of-band control can implement flexible communication at the microsecond level, meeting requirements for dynamic adjustment during AI training.

Table 5 lists the advantages, disadvantages, and application scenarios of the four control proposal. In addition to device forms, control isolation must be considered to prevent optical switches from being controlled by malicious NPUs, which may interrupt normal communication. These aspects will be continuously improved in subsequent work.

4.3.2 Optical Switch Configuration Process

P2P communication, the most basic communication mode, serves as the basis to implement collective communication for data exchange in AI training. Therefore, accurately and efficiently configuring an optical switch during training has become a key issue. Figure 16 shows a complete point-to-point communication configuration process in an optical interconnection architecture.

In Figure 16, there are two compute nodes: Node0 and Node1. Each compute node has two boards. One is a compute board that carries computing devices, such as CPUs and NPUs. The other is an optical switch board that carries lasers, sub-µs fast optical cross-connect components, and a general control unit. Components on the optical switch board are interconnected with the compute board through PCIe ports and are presented as PCIe devices to compute nodes. Compute nodes are connected through a MEMS OXC. An optical switch (1 x 16 at the transmit end and 16 x 16 at the receive end) is the most critical part of a fast optical cross-connect component, and the control plane of the fast optical cross-connect component mainly configures the optical switch on this component.

This section uses a scenario in which NPU4 and NPU5 in Node0 send data to NPU5 and NPU6 in Node1 respectively as an example to describe a point-to-point communication process based on a sub-µs fast optical cross-connect device, as shown in Figure 16. Numbers with red circles in Figure 16 represent the IDs of configuration steps. The complete point-to-point communication process is as follows:

1 The NPU4 at the transmit end inserts the command for switching the 1 x 16 optical switch before communication starts. After this command is successfully executed, NPU4 enters the waiting state.

2 The command is transmitted through PCIe in P2P mode to an optical switch board connected to the wavelength cross-connect component corresponding to Node0, and is written into a 1 x 16 optical switch corresponding to NPU4 (a corresponding register entry is written).

3 At the transmit end, the output port of the optical switch is switched to the MUX connected to Node1.

At the receive end, the preceding three steps are also performed to switch the input port of the optical switch to the DEMUX connected to Node0 and the output port of the optical switch to NPU5.

4 The link is re-established at the physical layer.

5 The link is re-established at the physical coding sublayer (PCS).

6 The network port of NPU4 on Node0 detects that the link is successfully established, stops waiting, and starts RDMA communication.

Before switching the optical switch, queue pair (QP) connections must be established between the RDMA network ports of all NPUs that need to communicate with each other. The link establishment process can be implemented by using other external networks. Alternatively, it can be implemented after the links that need to communicate are switched one by one using the wavelength cross-connect component before a tenant or task is allocated. Neither of the two implementation methods affects the system overall performance.

In addition, the optical switch of the wavelength cross-connect component must be correctly configured in the NPU communication process. Therefore, each NPU at the transmit end can access only the corresponding optical switch. During the configuration process, the configuration source needs to be identified and incorrect configurations need to be filtered out. At the receive end, because each NPU may be configured with any optical switch, permission control on a path needs to be maintained in the 16 x 16 optical switch. Once a path is occupied by an NPU at the

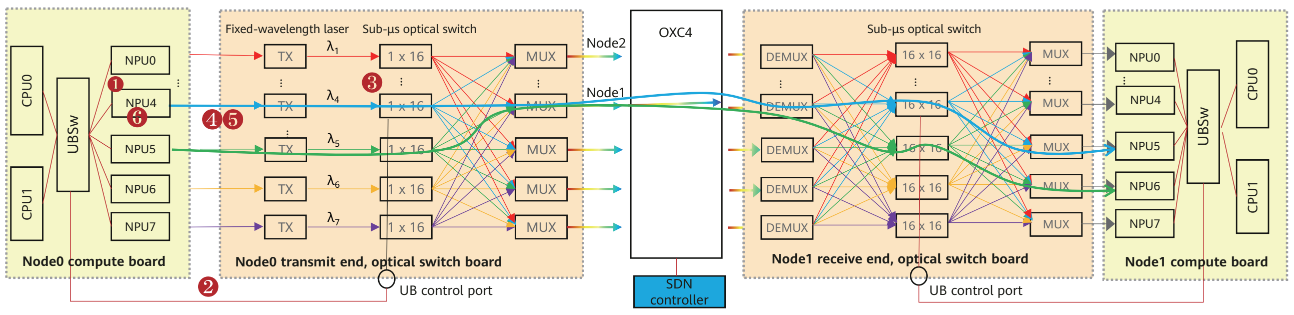


Figure 16 Point-to-point communication process of AI optical training

receive end, it cannot be configured for other NPUs. In this case, the out-of-band authentication code can be used for permission control.

This proposal brings an additional advantage: the connectivity of an optical path is determined by the algorithms of the source and destination ends. It is a distributed mechanism and does not require a centralized SDN controller, preventing single points of failures and performance bottlenecks in hotspots.

4.4 Topology-Aware Collective Communication Algorithms

The distributed training framework generally uses collective communication libraries to simplify model development. Such a library can abstract regular communication processes between multiple NPUs (e.g., AllReduce and AlltoAll). Compared with traditional point-to-point communication, the collective communication library greatly simplifies the development of large models. Examples of such libraries include open-source Open MPI, NCCL developed by NVIDIA, and HCCL developed by Huawei.

For the interconnection topology of AI optical training, we have designed topology-aware hierarchical collective communication algorithms to adapt to the full-mesh networking of optical interconnection and resolve the problem that optical interconnection cannot support multiple-to-multiple mappings concurrently.

4.4.1 Topology-Aware Hierarchical AllReduce Algorithm

An AI cluster with four nodes, each of which has four NPUs, is used as an example to describe the AllReduce communication process involving all NPUs (global communicator). In a conventional electrical switching scenario, the AllReduce operation in a global communicator can be divided into three steps, as shown in Figure 17.

Step 1: Intra-node ReduceScatter. The data of each NPU on a node is divided into four parts (marked A through D) based on the number of NPUs on the node. After the reduction operation is performed on each part of data, the data is scattered to each NPU.

Step 2: Inter-node plane-specific AllReduce. NPUs in the same position on different nodes form a plane. The AllReduce operation is performed on all NPUs on each plane. Take the plane formed by NPUs 0 as an example. All NPUs 0 (i.e., NPU 0, NPU 4, NPU 8, and NPU 12) of all nodes form plane 0. The Halving-Doubling AllReduce method performs the AllReduce operation on all NPUs in plane 0. The entire process requires two iterations: In the first iteration, the AllReduce operation is performed on adjacent nodes. After the operation is complete, both NPU 0 and NPU 4 have data A0 to A7, and both NPU 8 and NPU 12 have data A8 to A15. In the second iteration, the AllReduce operation is performed on the two nodes whose interval is 1. After the operation is complete, all NPUs have complete data A0 to A15.

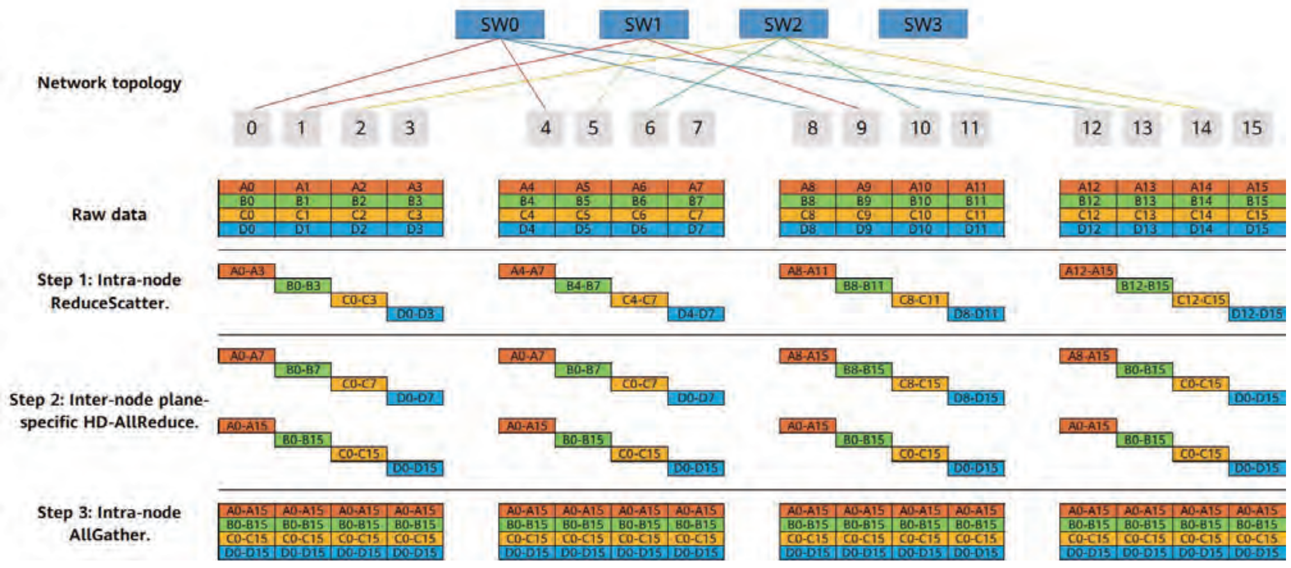
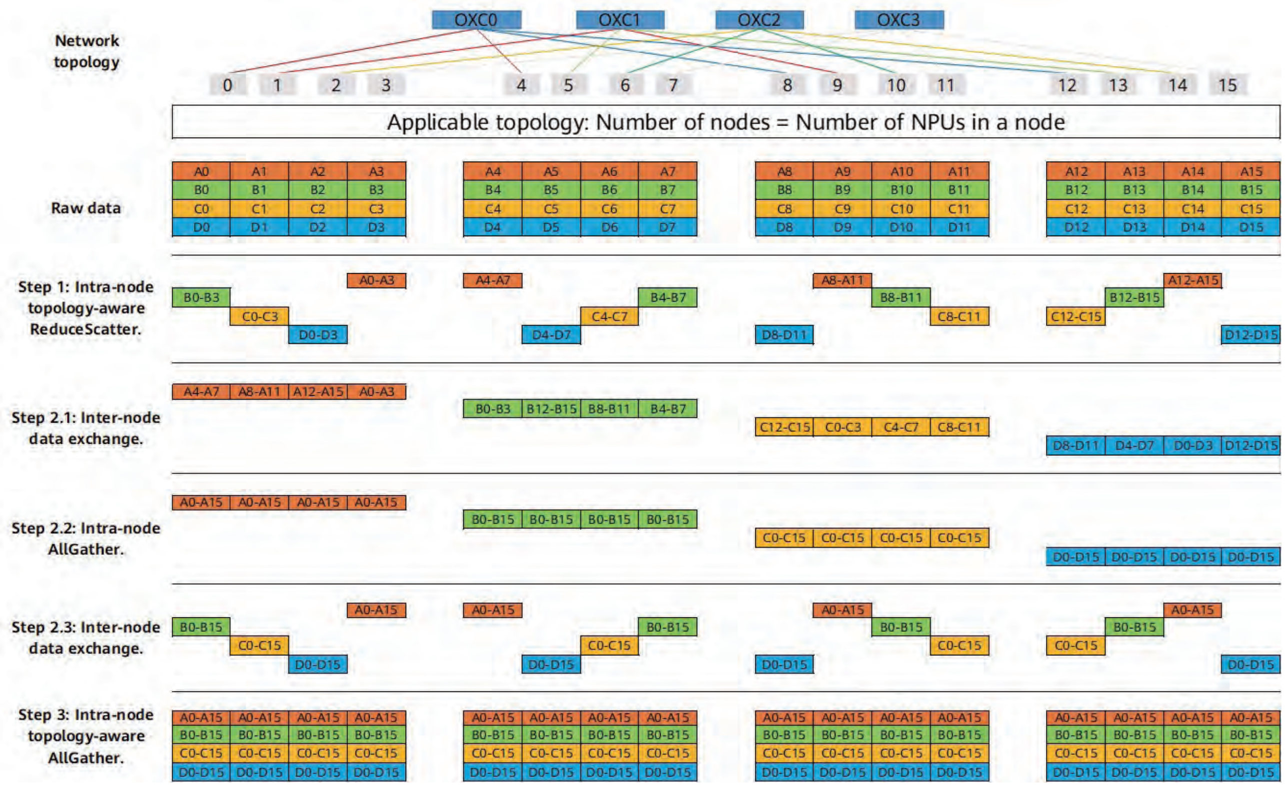


Figure 17 HD-AllReduce algorithm in an electrical switching scenario



Network topology
OXC0
OXC1
OXC2
OXC3
Applicable topology: Number of nodes = Number of NPUs in a node
Raw data
Step 1: Intra-node topology-aware ReduceScatter.
Step 2.1: Inter-node data exchange.
Step 2.2: Intra-node AllGather.
Step 2.3: Inter-node data exchange.
Step 3: Intra-node topology-aware AllGather.

Figure 18 HD-AllReduce algorithm in an optical switching scenario

Step 3: Intra-node AllGather. After the previous step, each node has complete data, but the data is distributed on different NPUs. Therefore, the AllGather operation is required to aggregate all data. After this operation is complete, all NPUs have the complete resultant data.

In an optical switching networking scenario, we have designed a topology-aware hierarchical AllReduce algorithm. As shown in Figure 18, the entire communication process can be divided into the following steps:

Step 1: Intra-node topology-aware ReduceScatter. The data of each NPU on a node is divided into four parts (marked A through D) based on the number of NPUs on the node. Unlike the ReduceScatter operation of electrical switching, the ReduceScatter operation of optical switching cannot ensure the full-mesh interconnection of all NPUs on the same plane. Therefore, each part of data is placed on an NPU that is directly connected to other nodes, so that the following steps can directly exchange data between nodes.

Step 2.1: Inter-node data exchange. Each NPU sends the data scattered to itself in the first step to the connected peer node and receives data from the peer node. After this step, each node has a part of data from A to D and this data scattered to different NPUs.

Step 2.2: Intra-node AllGather. The AllGather operation is performed on data in each node. After the operation is complete, the data of each NPU in the node is consistent.

Step 2.3: Inter-node data exchange. Each NPU sends its data to the connected peer node and receives data from the peer node. After the previous step is complete, the data of each NPU on the same node is consistent. Once the data exchange is complete, each node has the complete resultant data, but the data is distributed on different NPUs.

Step 3: Intra-node topology-aware AllGather. By performing the AllGather operation on the data on different NPUs in the node, each NPU can have the complete resultant data.

In addition, in the case of a non-power-of-2 number of nodes and a non-power-of-2 number of NPUs in each node, the topology-aware hierarchical AllReduce algorithm only needs to adjust the number of data parts and logical IDs (of topology awareness). Based on this algorithm, we have implemented and verified the system prototype described in Section 5.1, and completed the end-to-end training of the ResNet50 model.

For details about the quantitative performance comparison between this algorithm and the AllReduce algorithm on an electrical network, see Section 6.1.

4.4.2 Topology-Aware Hierarchical AlltoAll Algorithm

In large-scale AlltoAll communication, the industry typically uses the pair-wise algorithm. Figure 19 shows the AlltoAll algorithm based on pair-wise. This algorithm implements data exchange between NPUs through multiple steps. In each step, each NPU sends and receives data to fully utilize bidirectional link bandwidth.

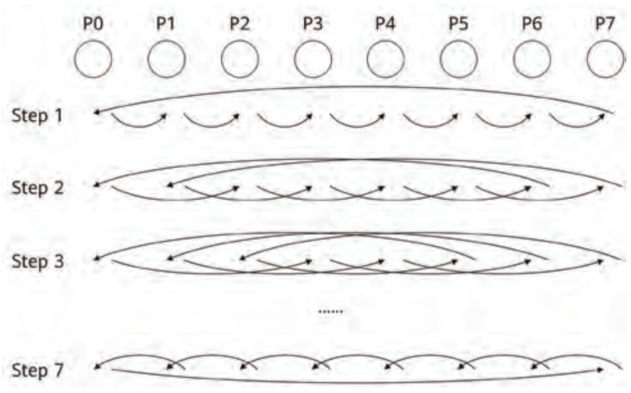


Figure 19 AlltoAll algorithm based on pair-wise

The number of NPUs is denoted as N, and the NPU ID is denoted as P_n, where $n \in [0, N-1]$. The step ID starts from 1, that is, $step \geq 1$. In this case, the pair-wise algorithm can be expressed as follows: In each step, NPU P_n receives data from NPU P_{N+step}, and sends the data to NPU P_{N-step}. Modular addition and subtraction methods are used here. If overflow or underflow occurs, the NPU ID will start from the smallest or largest NPU ID. For example, P_{-1} is P_{N-1}, and P_{PNUM} is P_0.

However, in the optical interconnection architecture, each NPU in the pair-wise algorithm needs to receive data from one node and send data to another node. The multiple-to-one communication pattern does not match the point-to-point optical interconnection characteristic, as shown in Figure 20.

This paper proposes a topology-aware hierarchical AlltoAll algorithm — an adaptation of the pair-wise algorithm to optical interconnection communication — to ensure that

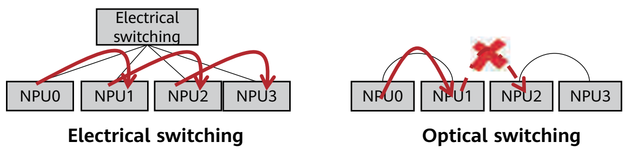


Figure 20 Pair-wise algorithm not applicable to the optical interconnection architecture

each NPU between nodes only needs to communicate with another NPU. An AI cluster with four nodes, each of which has four NPUs, is used as an example to describe the AlltoAll communication process involving all NPUs (global communicator), as shown in Figure 21. The topology-aware hierarchical AlltoAll algorithm in optical interconnection scenarios can be divided into three steps.

Step 1: Intra-node topology-aware AlltoAll. Data on each NPU in a node is divided based on the number of NPUs involved in AlltoAll communication. In this example, data on each NPU is divided into 16 parts, which are marked as A to P. Because there are only four NPUs on a node, only four parts of data are required to implement the intra-node AlltoAll operation. Therefore, 16 parts of data are combined into four groups. After this step is completed, four groups of data on each NPU within a node are scattered on four different NPUs.

Step 2: Inter-node data exchange. In an optical switching scenario, each node has only one NPU directly connected to another node, and the connected NPU sends all the four parts of data to the corresponding NPU. After this step is completed, 16 parts of data on each NPU are distributed to all nodes.

Step 3: Intra-node topology-aware AlltoAll. In this step, the 16 parts of data required by each NPU are scattered on other NPUs (unlike in step 1, where the data was divided into four groups). During the intra-node AlltoAll operation, the offset of each part of data in the buffers of the transmitter and receiver needs to be adjusted. After this step is completed, data exchange is completed between the 16 NPUs of the global communicator.

In addition, in the case of a non-power-of-2 number of nodes and a non-power-of-2 number of NPUs in each node, the topology-aware hierarchical AlltoAll algorithm only needs to adjust the number of data parts and logical IDs (of topology awareness). Note that if a fast optical cross-connect component is used, only the first two steps are required for the hierarchical AlltoAll algorithm. In the second step, when the data arrives at the peer node, only the fast optical cross-connect component needs to be used to switch the destination in the node to the corresponding NPU so that the communication efficiency of the AlltoAll operator can be further improved.

For details about the performance comparison between this algorithm and the AlltoAll algorithm on an electrical network, see Section 6.2.

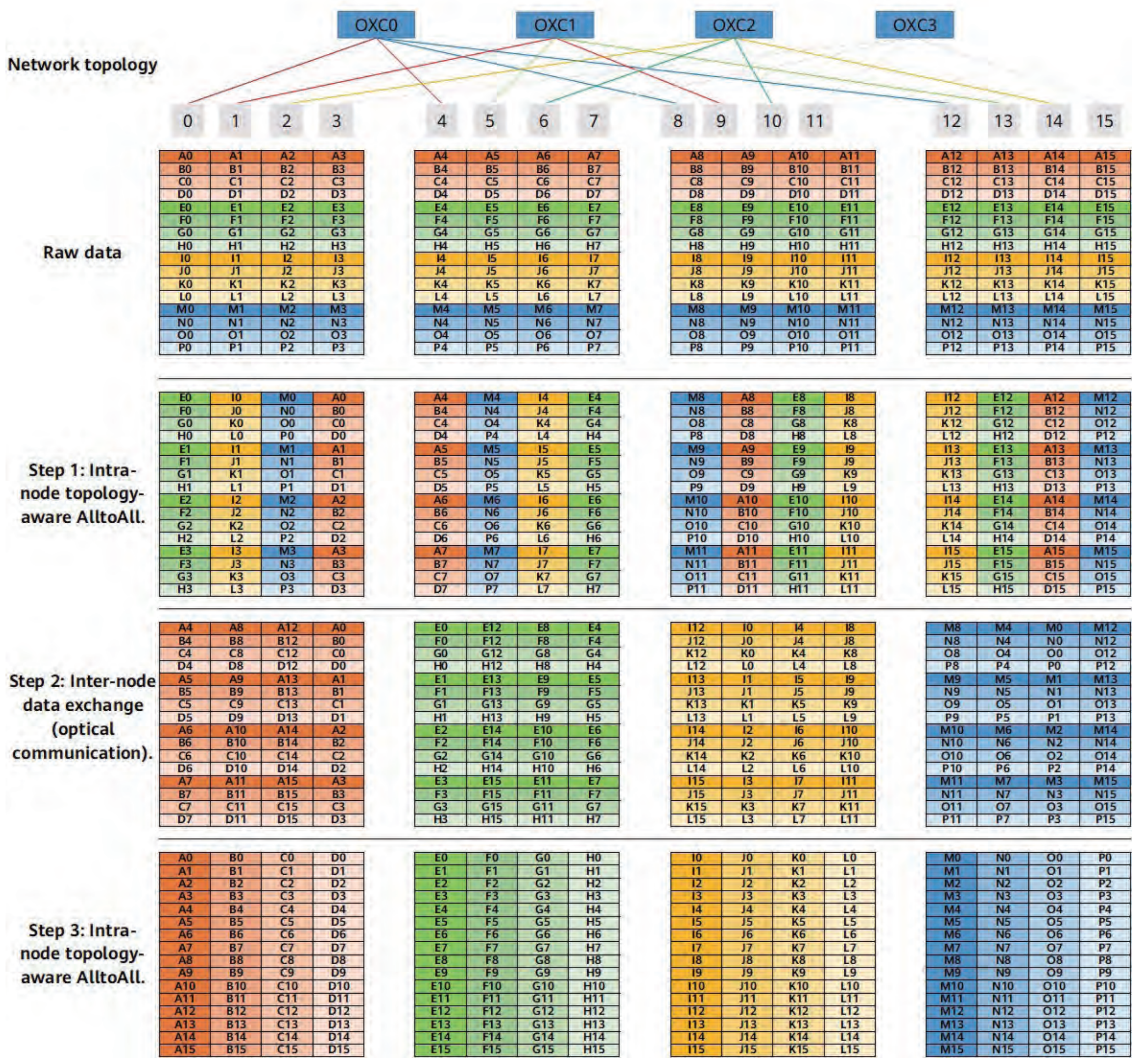

OXC0
OXC1
OXC2
OXC3
Network topology
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Raw data
Step 1: Intra-node topology-aware AlltoAll.
Step 2: Inter-node data exchange (optical communication).
Step 3: Intra-node topology-aware AlltoAll.

Figure 21 Topology-aware hierarchical AlltoAll algorithm in optical interconnection scenarios

4.5 Topology-Aware Parallelism Policy

In addition to collective communication algorithms, optical interconnection technologies need to be adapted. The parallelism policy directly determines optical link scheduling and has a greater impact on training performance. Therefore, such a policy needs to be optimized for optical scenarios. This section uses the Pangu-α model as an example to describe the optimization policy of the pipeline parallelism model for optical interconnection scenarios.

The Send/Recv operator mainly exists in the pipeline parallelism mode, as shown in Table 2. The main sources of the Send operation in Pangu-α pipeline parallelism are as follows:

1 A stage sends the feature map to the next stage.

2 The next stage sends gradient data to its previous stage.

3 The first stage sends the embedding table to the last stage. The Recv operation is a reverse communication operator corresponding to the Send operation.

In an optical communication scenario, a connection may not be established between an NPU in a stage and an NPU in a next stage. Therefore, the Send operation needs to be adapted to optical interconnection scenarios.

Assume that Pangu-α runs in pipeline parallelism mode in an environment with 8 servers and 64 NPUs. Figure 22 shows the Send/Recv communication modification pattern. In the Pangu-α model, the AllReduce operation has

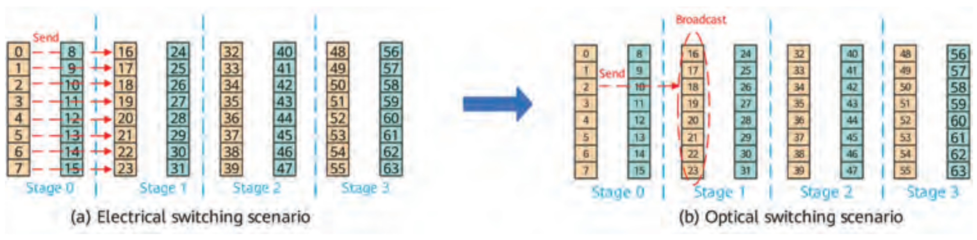

Send
Stage 0
Stage 1
Stage 2
Stage 3
(a) Electrical switching scenario
Broadcast
Send
(b) Optical switching scenario

Figure 22 Adapting Send/Recv communication to optical switching scenarios

been performed for eight NPUs in a node before all Send operations. Therefore, before each NPU performs the Send operation, the data to be sent by all NPUs is the same. Only one NPU is required to perform the Send operation. The selected NPU is the one connected between two stages. After reaching the NPU of the next stage through the Send operation, the data is broadcast to all NPUs within the node through the Broadcast operation in the node.

In some models, no AllReduce operation is performed before the Send operation, and the data sent by each NPU is inconsistent. In this case, the intra-node forwarding process is added to the implementation of the Send/Recv algorithm. Assume that NPU 0 performs the Send operation to send data to NPU 16. The two stages have only NPU 2 connected between them. The data of NPU 0 needs to first be sent to NPU 2 of this node. Then, NPU 2 sends the data to NPU 18 of the next stage, and NPU 18 forwards the data to NPU 16 of the destination node.

The two Send operations do not conflict with each other. They are dynamically adjusted based on the model configuration and network topology during running. The topology-aware model parallelism policy can better optimize the training performance of large models.

5 System Prototype of AI Optical Training

The current prototype system uses a MEMS OXC to build an all-optical AI training cluster with 8 servers and 64 NPUs. It has tested ResNet50 in two network architectures: all-optical interconnection and all-electrical interconnection.

5.1 System Prototype

To verify the low-latency AI optical interconnection proposed in this paper, we have set up a small prototype system, as shown in Figure 23. Currently, no stable prototype is available for wavelength aggregation optical components, but MEMS OXC products are relatively mature. Therefore, the prototype system contains only Huawei-developed MEMS OXC. The lack of fast optical cross-connect components makes it difficult to verify ultra-large-scale AI applications. However, optical interconnection networks can still be quickly tested on the current mainstream large model.

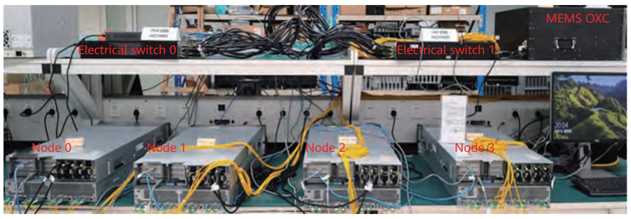


Figure 23 Prototype of an 8-server 64-NPU AI optical interconnection cluster

Figure 24 shows the framework of a prototype system. The AI cluster system consists of eight nodes, each of which is a Huawei-developed Atlas 800-9010 server. Each node contains eight Ascend 910B NPUs, meaning there is a total of 64 NPUs. Each NPU provides a network port to connect to the MEMS OXC, which itself is connected to the fat-tree electrical network formed by two layers of electrical switches.

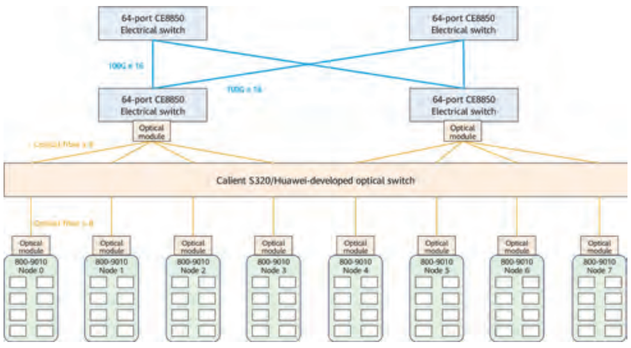


Figure 24 Prototype framework

Such a networking architecture enables the prototype system to test the performance of AI applications in both optical and electrical networking environments. To perform the test in an optical interconnection network, the optical switch connection policy of the MEMS OXC can be configured to bypass electrical switches in the fat-tree structure. This allows all NPUs to communicate with each other through the MEMS OXC. To perform the test in an electrical interconnection network, the MEMS OXC can be configured to provide optical ports, and NPUs directly communicate with each other through the two-layer fat-tree electrical network. Due to an ultra-low transmission latency of the MEMS OXC, latency overheads caused by the MEMS OXC in the electrical interconnection network can be ignored.

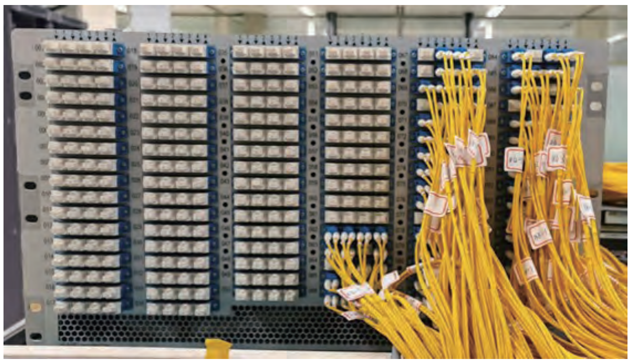


Figure 25 Huawei-developed 400 x 400 3D MEMS OXC prototype

Figure 25 shows the prototype of Huawei-developed 3D MEMS OXC. This optical switching prototype is equipped with 400 pairs of available optical ports. In addition, it features low insertion loss (lower than 3.0 dB) and ultra-low transmission latency (lower than 30 ns for all-optical interconnection transmission). Currently, however, the configuration latency is still high. It takes about 1000 ms to configure optical switches for all ports. The related research team is working to lower this configuration latency for the 512-port MEMS OXC.

5.2 Test Results

With the prototype system described earlier, we have tested the ResNet50 model under both the optical interconnection network and electrical interconnection network. In the test, the ImageNet 2012 training dataset with a sample batch size of 256 is used. The prototype system tests ResNet50 in the fat-tree electrical interconnection environment and all-optical interconnection environment. Table 6 lists the test data.

The main collective communication operator in ResNet50 is AllReduce. Although AllReduce has been adapted to optical interconnection networks, the average latency of the AllReduce operator in such networks is still about 13.4% higher than that of electrical interconnection networks. The main causes are as follows: (1) To adapt to all-optical interconnection clusters, the AllReduce algorithm for optical interconnection has two extra inter-node data copying operations. (2) Currently, the topology of eight NPUs in the Atlas 800-9010 server is 2 x 4 NPU mesh, meaning that the HD AllReduce algorithm cannot be used. Therefore, compared with step 2 in the algorithm for electrical interconnection, step 3 in the AllReduce algorithm for optical interconnection has extra startup latency. In the next-generation Ascend server (1980B or 1981), the topology between NPUs in a node is the full mesh of eight NPUs. This means that the HD algorithm can be used in step 3 of the AllReduce algorithm for optical interconnection, eliminating the introduction of additional startup latency.

Because communication latency can be overlapped by computation in a training process, the performance

Table 6 ResNet50 test results

Latency (ms)		Electrical Switching	Optical Switching	Impact on Latency
Average latency of a single iteration		162.9853	163.0159	+0.019%
Forward + reverse latency		161.1194	161.1569	+0.023%
Average latency of AllReduce	Average latency of AllReduce1	5.6878	6.3994	+12.5%
	Average latency of AllReduce2	1.0341	1.2829	+24.06%
	AllReduce communication latency that cannot be overlapped with computation	1.548	1.6968	+9.6%
	Overall latency	8.2699	9.3791	+13.4%

deterioration of model training in an optical interconnection scenario is far less significant than that of an individual AllReduce operator. In ResNet50, an AllReduce operator is usually split into multiple AllReduce sub-operators. For example, Table 6 shows that the AllReduce operator is split into sub-operators AllReduce1 and AllReduce2 in this test. After some results are computed, AllReduce1 can be started and completely overlapped by the computation phase. In terms of end-to-end latency, the performance of ResNet50 in all-optical interconnection network scenarios is close to that of ResNet50 in collision-free pure electrical network scenarios. The performance difference is only 0.019%.

The purpose of the preceding phased work is to demonstrate the feasibility of optical interconnection with limited labor by using the current combination of software and hardware. Therefore, optimizing performance is not a top priority. For the measures of further performance improvement, see Section 6.1.

5.3 Power Consumption Analysis

The next-generation chip and hardware platform is used as an example to estimate the power consumption of a network with 32K (K = 1024) NPUs.

There are four major items:

1 Super node: It includes all scale-up interconnections and the L1 and L2 of a scale-out network. The two parts are the same in the optical and electrical networks.

2 AI parameter plane network: It uses the traditional fat-tree structure in an electrical network, including many optical modules from super nodes to core switches and 768-port modular switches. Replacing the modular switches with OXC devices reduces the number of required optical modules by half and the end-to-end system power consumption by 7%.

3 Service plane network: It is a low-bandwidth network that serves management, control, and storage access. This network is the same in optical and electrical networks.

4 Storage: The optical and electrical networks share the same storage.

In general, replacing large modular switches with MEMS OXC devices achieves a 7% reduction in power consumption.

6 Future Work

To give full play to the potential of optical interconnection technologies, further exploration is needed in terms of scenarios, optical components, and corresponding algorithms.

6.1 Pipelined Communication Latency Overlapping AllReduce Algorithm

As shown in Table 7, the performance of the all-optical algorithm in Figure 18 is lower than that of the pure electrical algorithm in Figure 17.

Where:

1 *S* indicates the data volume on a single NPU.

2 *N* indicates the number of NPUs on a single node or super node. To simplify the discussion, it is assumed that the number of nodes or super nodes is also *N*.

3 *BI* indicates the intra-node bandwidth.

4 *BO* is the inter-node bandwidth. Generally, *BI* is much higher than *BO* by 5 to 10 times .

Table 7 Power consumption analysis

		AI Electrical Interconnection					**AI Optical Interconnection**				
				32K scale					32K scale		
Calculation (including L1 and L2)	Super node (including L1 and L2 switching)		220	256	56320	56320	Super node (including L1 and L2 switching)	220	256	56320	56320
Network	AI parameter plane network	AI plane-L2-L3 optical module 400G	0.01	131072	1310.72	5292.94222	AI plane-L2-L3 optical module 400G	0.01	65536	655.36	688.128
		AI plane-modular switches with 768 ports (only 576 ports are used)	35	113.777778	3982.22222		AI plane-MEMS OXC with 512 ports	0.256	128	32.768	
	Service plane network (including storage network)	Service plane-L2-L3 optical module 200G	0.004	68812.8	275.2512	1320.58453					
		Service plane-modular switches with 768 ports (only 576 ports are used)	35	29.8666667	1045.33333						
Storage		Network ports on the storage device 100D: 1/20	1.1	819.2	901.12	901.12					
						63834.65					**59229.83**
										Power consumption reduction ratio	**0.072137**
										kW/NPU	**0.1439**

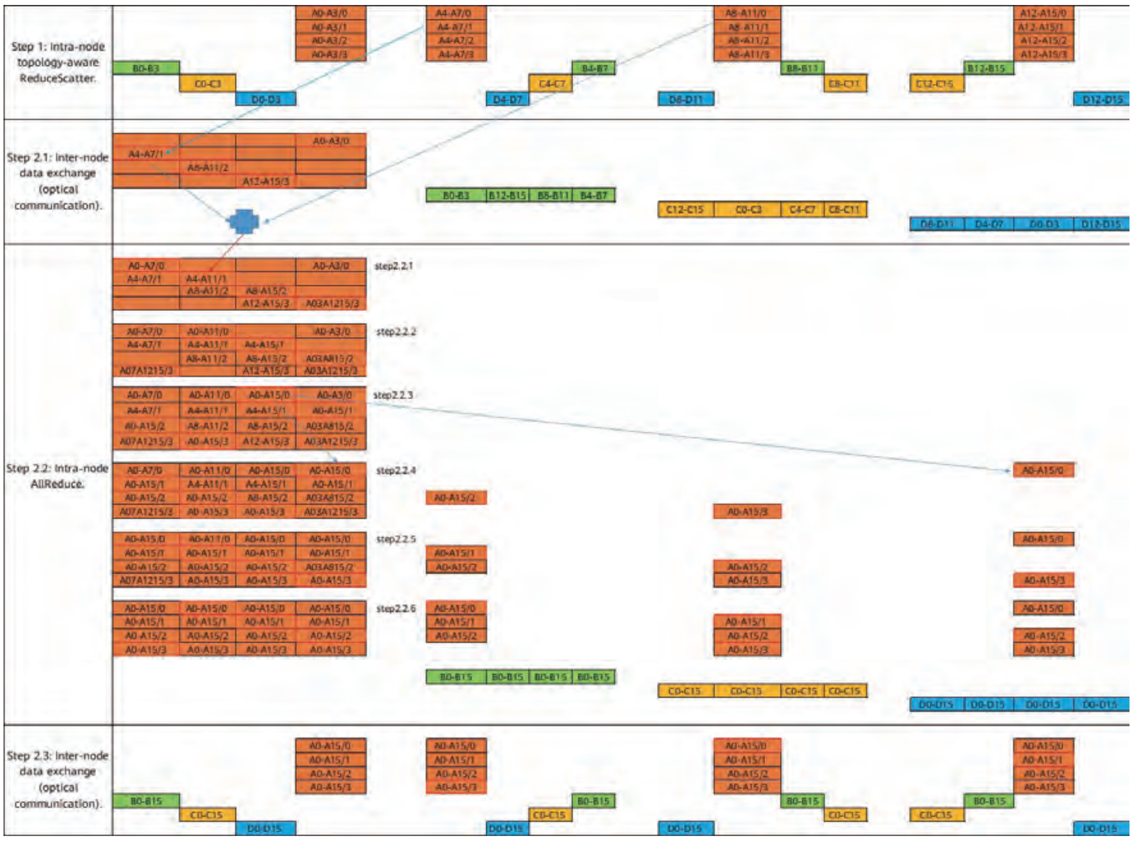
Table 8 Comparison between optical and electrical AllReduce algorithms

	Electrical Network AllReduce in Figure 17	Optical Interconnection AllReduce in Figure 18	Pipelined AllReduce in Figure 26
Step 1: ReduceScatter.	$\frac{N-1}{N} * \frac{S}{BI}$	$\frac{N-1}{N} * \frac{S}{BI}$	$\frac{N-1}{N} * \frac{S}{BI}$
Step 2.1: Inter-node copying.		$\frac{S/N}{BO}$	0
Step 2.2: Intra-node ring AllReduce.	$2 * \frac{N-1}{N} * \frac{S/N}{BO}$	$2 * \frac{N-1}{N} * \frac{S/N}{BI}$	$2 * \frac{N-1}{N} * \frac{S/N}{BO}$
Step 2.3: Inter-node copying.		$\frac{S/N}{BO}$	0
Step 3: AllGather.	$\frac{N-1}{N} * \frac{S}{BI}$	$\frac{N-1}{N} * \frac{S}{BI}$	$\frac{N-1}{N} * \frac{S}{BI}$

As shown in Table 8, step 1 and step 3 of the two algorithms are the same. Considering that $\frac{N-1}{N} \approx 1$ when *N* is relatively large in a case of a super node [45], only $2 * \frac{S/N}{BI}$ in step 2.2 is added to the optical interconnection algorithm in Figure 17.

In addition, considering that the intra-node bandwidth *BI* is much higher than the inter-node bandwidth *BO*, step 2.2 can be covered in steps 2.1 and 2.3, as shown in the following figure.

The algorithm in Figure 26 divides each piece of data in Figure 18 into four sub-pieces. For example, A4-A7 in row 10 is divided into A4-A7/0, A4-A7/1, A4-A7/2, and A4-A7/3. In addition, the first half of ring AllReduce in step 2.2 is expanded into six sub-steps: 2.2.1 to 2.2.6. The red box of each sub-step represents a sub-piece of data copied from another node. For example, A4-A7/1 of NPU 0 on node 0 in row 20 is copied from NPU 0 on node 1 in row 11. The time required for each copying operation is $\frac{1}{N} * \frac{S/N}{BO}$.



Step 1: Intra-node topology-aware ReduceScatter.
Step 2.1: Inter-node data exchange (optical communication).
Step 2.2: Intra-node AllReduce.
Step 2.3: Inter-node data exchange (optical communication).
step2.2.1
step2.2.2
step2.2.3
step2.2.4
step2.2.5
step2.2.6
B0-B3
C0-C3
D0-D3
B0-B15
C0-C15
D0-D15

Figure 26 Pipelined AllReduce algorithm

From sub-steps 2.2.1 to 2.2.3, each red box indicates a sub-piece of data with a size of S/N^2 and which is copied from the NPU that is directly connected on another node. In addition, a sub-piece of data with the same size is copied from the adjacent NPU on the same node, and the two sub-pieces of data are accumulated to obtain a new partial accumulation result. Considering that the inter-node bandwidth *BO* is far lower than the intra-node bandwidth *BI* and memory bandwidth, the time of each sub-step is S/(N^2 x BO).

From sub-steps 2.2.4 to 2.2.6, each red box indicates a sub-piece of data with a size of S/N^2 and which is distributed between nodes or within a node. Similarly, considering that the inter-node bandwidth *BO* is far lower than the intra-node bandwidth *BI* and memory bandwidth, the time of each sub-step is S/(N^2 x BO).

Therefore, the latency overhead of step 2 in the algorithm in Figure 26 is 2 x (N – 1) x S/(N^2 x BO). The performance is the same as that of the pure electrical algorithm in Figure 17 without hash collisions.

6.2 Pipelined Communication Latency Overlapping AlltoAll Algorithm

As shown in Table 9, the performance of the all-optical algorithm in Figure 17 is lower than that of the pure electrical algorithm in Figure 16.

Considering that large super nodes have become the trend in the AI field [45], *N* generally reaches 256 or even 384. In this case, $\frac{N-1}{N} \approx 1$. Considering that the intra-node bandwidth *BI* is 10 times higher than the out-of-node bandwidth *BO*, the running time of the optical interconnection AlltoAll in Figure 21 is about 1.2 x S/BO. The running time of the electrical network AlltoAll in Figure 19 is about S/BO without hash collisions. The optical interconnection AlltoAll is about 20% slower than the electrical network AlltoAll.

Similar to algorithms in the previous section, the pipelined AlltoAll algorithm can also be used to cover the extra latency of steps 1 and 3 of the optical interconnection AlltoAll algorithm in Figure 21 into step 2. As shown in Figure 27:

1 On the node where NPUs 0 to 3 reside, step 1 of the optical interconnection AlltoAll algorithm attempts to collect all the blue data (that needs to be sent to the node where NPUs 12 to 15 reside) to NPU 2, so that step 2 can use a yellow optical fiber to send the data to NPU 14 in one go. However, in the original data, the M2 to P2 data in the red box already exists on NPU 2 and no wait time is consumed. As shown by **(1)**, the M2 to P2 data can be directly sent to NPU 14, and the consumed time is $\frac{1}{N} * \frac{S}{BO}$.

2 Meanwhile, as shown by **(2)**, step 1 of the optical interconnection AlltoAll algorithm is executed to send the N0 to P0, N1 to P1, and N2 to P2 data in the three black boxes to NPU 2. Because the intra-node bandwidth *BI* is much higher than the inter-node bandwidth *BO*, this step is completed before the operation shown by **(1)**.

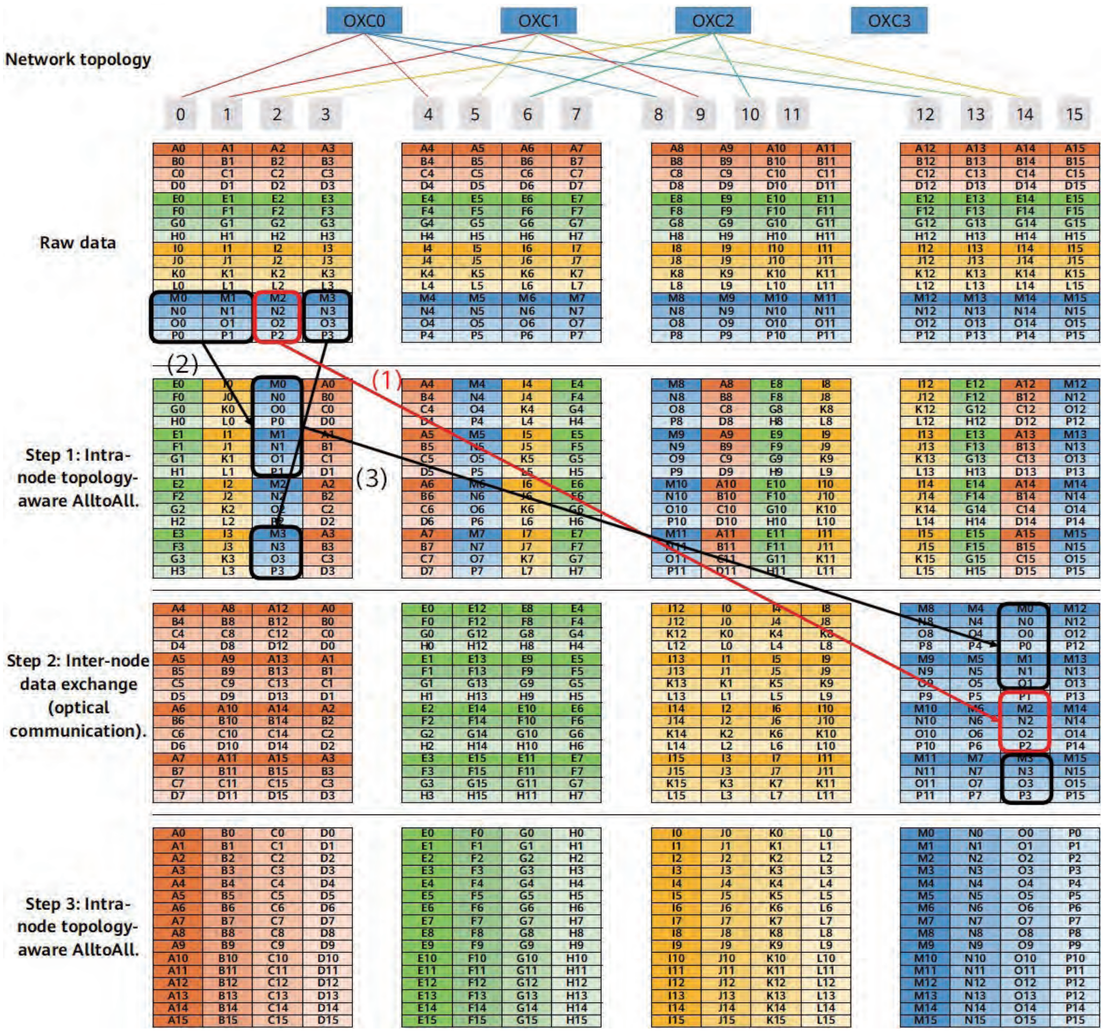
3 After the operation shown by **(1)** is complete, the N0 to P0, N1 to P1, and N2 to P2 data is sent to NPU 14, and the consumed time is $\frac{N-1}{N} * \frac{S}{BO}$.

In this way, the time of step 1 is covered by step 2. Similarly, the time of step 3 can also be covered by step 2 through the pipeline. Therefore, the actual performance of the optical interconnection AlltoAll can be equivalent to that of the electrical network AlltoAll in Figure 19 without hash collisions.

However, hash collisions cannot be avoided on an electrical network with more than three layers. As such, the performance advantages of an optical network will be further reflected. Table 9 compares the performance of a single AlltoAll operator with the E2E performance in the Shennong model with 64 NPUs.

Table 9 Running time comparison between optical and electrical AlltoAll algorithms

	Electrical Network AlltoAll in Figure 19	Optical Interconnection AlltoAll in Figure 21	Pipelined AlltoAll in Figure 26
Step 1: Intra-node AlltoAll.		$\frac{N-1}{N} * \frac{S}{BI}$	
Step 2: Inter-node AlltoAll.	$\frac{N-1}{N} * \frac{S}{BO}$	$\frac{S}{BO}$	$\frac{S}{BO}$
Step 3: Intra-node AlltoAll.		$\frac{N-1}{N} * \frac{S}{BI}$	



OXC0
OXC1
OXC2
OXC3
Network topology
Raw data
Step 1: Intra-node topology-aware AlltoAll.
Step 2: Inter-node data exchange (optical communication).
Step 3: Intra-node topology-aware AlltoAll.
(1)
(2)
(3)

Figure 27 Pipelined AlltoAll algorithm

It is expected that the performance loss caused by hash collisions will reach about 30% on a large network with more than 4K (K = 1024) devices. The hybrid optical/electrical network will be able to compensate for this loss.

6.3 Ideas for HPC and Hybrid HPC/AI Scenarios

With the development of AI technologies, using AI to enable traditional HPC has become a new research direction. Similar to Pengcheng Cloud Brain Phase III, large-scale supercomputing infrastructure that emphasizes hybrid HPC/AI scenarios is also emerging. Applying optical interconnection technologies to enable such scenarios has also become a high-value exploration direction.

Unlike traffic in AI scenarios, HPC network traffic has the following characteristics:

1 In HPC applications, traffic is relatively irregular and lacks periodic characteristics, and traffic predictability is poor.

2 According to the statistics of current supercomputing centers [24], most of the traffic comes from small- and medium-scale jobs that use most of the system cores. Such jobs generally use one to four groups (nodes/

Table 10 Performance loss of the Shennong model with 64 NPUs

Scenario	No Traffic Conflict	A Few Traffic Conflicts at the Spine Node	Loss
Latency of a single AlltoAll (ms)	**27.78**	**17.10**	**38.4%**
Average latency of a single iteration (ms)	**17340.583**	**18056.241**	**4%**

super nodes) to achieve good coverage. Because jobs are scheduled on different nodes, the bandwidth traffic of links on the entire network is unbalanced, and only a few links are fully loaded [25].

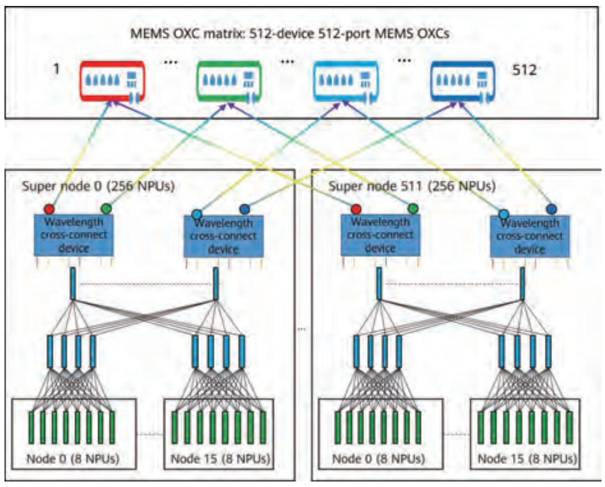

MEMS OXC matrix: 512-device 512-port MEMS OXCs
1
512
Super node 0 (256 NPUs)
Super node 511 (256 NPUs)
Wavelength cross-connect device
Node 0 (8 NPUs)
Node 15 (8 NPUs)

Figure 28 FAT-Dragonfly+ networking

Based on the traffic characteristics of the preceding HPC and hybrid HPC/AI scenarios, we have designed a reconfigurable FAT-Dragonfly+ topology, as shown in Figure 28. The characteristics of this architecture are as follows:

1 Two-layer fat-tree networking is used in the group to form a flexible high-bandwidth electrical interconnection area.

2 Outgoing switches in the group are connected to wavelength cross-connect components to implement µs-level fast and flexible adjustment of inter-group bandwidth. As shown in Figure 29a, under a specific tenant scale and job allocation, when a burst elephant flow exists between two super nodes, traffic of multiple wavelengths can be dynamically accommodated on one optical fiber and one port of the MEMS through a wavelength component. For details about the control plane process, see Section 4.3.

3 In addition, the large-capacity port switching OXC is used to isolate the inter-group traffic of tenants. As shown in Figure 29b, after tenants A and B are allocated, there is no traffic on the blue optical path between them. Corresponding optical modules can be configured for optical paths inside the tenants to improve the bandwidth between multiple super nodes inside the tenant. This helps overcome the problem of inherent top-layer bandwidth insufficiency in the Dragonfly+ topology.

This hierarchical network structure gives full play to the advantages of optical and electrical networks. It is an efficient hybrid optical-electrical architecture suitable for future HPC scenarios. Further exploration is needed in order to realize a feasible proposal with this architecture.

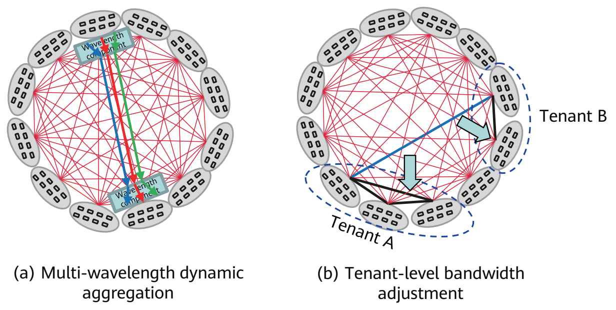

Wavelength component
Wavelength component
Tenant B
Tenant A
(a) Multi-wavelength dynamic aggregation
(b) Tenant-level bandwidth adjustment

Figure 29 Dynamic wavelength aggregation and tenant-level bandwidth adjustment

6.4 Large-Scale Modeling and Simulation System

Although the prototype system can verify the performance of this proposal on current Ascend training servers (Hi1980) in an E2E manner, there are still some limitations:

1 The next-generation Ascend chip is not mature, and using it to build a prototype system is difficult. Therefore, the performance of optical networking in the next-generation training cluster cannot be verified.

2 The networking scale of the prototype system is limited, and specifications such as performance, power consumption, and cost of a large-scale cluster cannot be effectively predicted.

3 The modification and adaptation of communication algorithms and software stacks are usually labor hungry, which cannot be quickly verified.

Therefore, in the absence of actual hardware, we plan to implement a large-scale modeling and simulation platform to better verify the communication and end-to-end performance of all-optical AI training proposed in this paper based on next-generation Ascend chips and large-scale (K-level or 10K-level, K = 1024) clusters. Currently, the industry has witnessed many large-scale simulation platform applications in AI scenarios. Table 11 lists mainstream simulation methods.

Based on the analysis of mainstream simulators, we have drawn the following conclusions:

1 A cycle-accurate (CA)-level simulator has high precision but a low simulation speed, whereas a polynomial simulator has a high simulation speed but low precision.

Table 11 Mainstream simulation methods

	CA-Level Simulation	Fine-Grained Cluster Simulation	Coarse-Grained Cluster Simulation	Polynomial-Cost Model
Typical simulator	GPGPU-SIM[11] Gem5 [26]	PANAMA [27] ASTRA-SIM [28]	SIP-ML [6] TripletRun [29]	Anonymous simulator [30]
Characteristics	Cycle level, with network model, operator/application/ communication traffic	Traffic model input, event-driven, with network model	Traffic model input, event-driven, without network model	No runtime phase, direct calculation of latency using formulas
Node size	32 NPUs or CPUs; up to 1K when distributed simulation is used	More than 1K; up to 80K when distributed simulation is used	80K or more	Cluster of any scale
Traffic input	Single-operator and pure communication traffic	Pure communication traffic and application task flow	Application task flow	Application task flow
Simulation granularity	Clock level	Packet and flit levels	Flow level	Operator level or higher
Simulation speed	Slow	Medium	Fast	Fastest
Implementation at the communication protocol layer	Yes	Yes	No	No
Implementation on the switch side	Yes	Yes	No network model is available, but traffic conflict modeling is available.	No
Accuracy	Accurate, basically the same as the actual test results	Accurate in the network part	Medium	Low
Application scenario	Single-chip application	HPC/AI	AI	AI

[11] GPGPU-SIM: http://www.gpgpu-sim.org/

Neither of them is suitable for performance simulation of a large-scale optical networking cluster.

2 Coarse-grained and fine-grained cluster simulations can be implemented simultaneously. The combination of the two simulation modes can satisfy the requirements of quick performance evaluation for large-scale clusters and accurately simulate the communication layer that we focus on. Coarse-grained cluster simulation can be used to quickly evaluate large-scale overall performance (when the accuracy meets the requirement), and fine-grained cluster simulation can be used to evaluate the communication microarchitecture's impact on overall performance.

3 The simulator's capability of accurately evaluating the communication layer requires that the traffic input be compatible with actual application task flows and pure communication traffic.

Based on the networking models, communication traffic, and chip capabilities in the mentioned AI scenarios, the objectives of a large-scale simulation system are as follows:

1 A simulation platform that includes integrated intra-node and inter-node network models needs to be built to support electrical and optical networking architectures, current and next-generation Ascend chips and hardware capabilities, as well as large-scale simulation topologies at different node sizes.

2 The simulator supports the implementation of collective communication operators in AI scenarios, including AlltoAll, AllReduce, AllGather, and Broadcast.

3 The simulator supports interconnection with the input of real AI models and implementation of real AI models in distributed training mode.

4 The simulator can be compatible with the simulation of both fine-grained and coarse-grained communication models. Fine-grained simulation reflects the impact of communication protocol features and chip capabilities on overall performance, and coarse-grained simulation is used for fast simulation in large-scale networking.

5 The fine-grained simulation model supports both intra-node direct memory access (DMA) communication

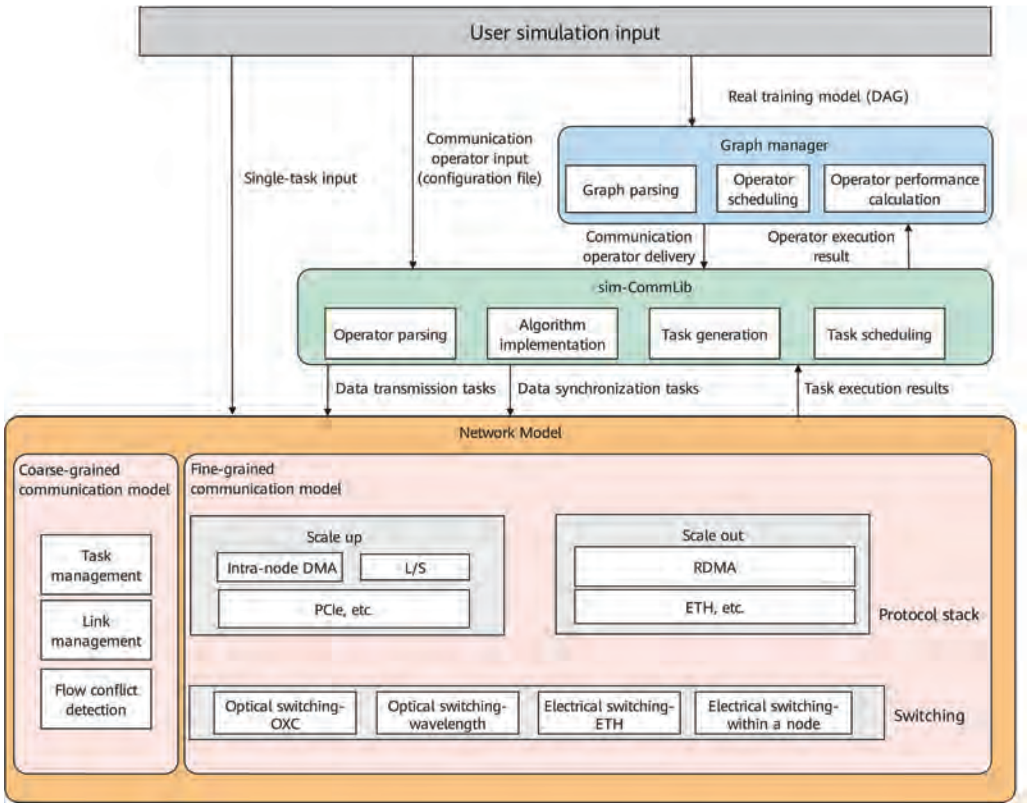


Figure 30 Large-scale modeling and simulation platform

and inter-node remote direct memory access (RDMA) communication, as well as communication protocols corresponding to the two communication modes.

The simulator is triggered based on events and supports three levels of inputs: real training model, single communication operator, and single task. Figure 30 shows the overall architecture of a simulation system, which is mainly comprised of the following three modules:

1 **Graph manager**: It parses the data flow diagram of the real AI training model, and schedules and executes operators in the diagram. Communication operators are delivered to the sim-CommLib module, and the input of compute operators comes from the actual test results or ESL operator simulation results.

2 **sim-CommLib**: It parses communication operators in the communication operator libraries (e.g., HCCL, NCCL, and MPI) and parses these communication operators into tasks. The generated tasks include data transmission tasks and data synchronization tasks. This module delivers the two types of tasks to the Network Model module based on the configured scheduling policy.

3 **Network Model**: It implements the network model, receives and executes communication tasks, and returns the task execution results to the sim-CommLib module. The Network Model module supports the implementation of both fine-grained and coarse-grained communication models. The fine-grained communication model can simulate the completion time of each task after being processed by the communication protocol stack and the switch side. The coarse-grained communication model focuses on traffic conflicts, but does not implement protocol stack details.

7 Related Work

There are two main directions to organize AI clusters. One is the hierarchical architecture of NVIDIA [45] and Huawei. With this architecture, ultra-high bandwidth is provided in nodes and super nodes, and hierarchical AllReduce and AlltoAll algorithms are designed based on the locality of communication to maximize the advantages of high bandwidth and low latency in nodes. This paper follows this direction. The other direction is an architecture similar to the Dojo cluster of Tesla, which forms a two-dimensional mesh topology and lacks obvious hierarchical features. This architecture is rooted in Tesla's own service characteristics. That is, Tesla mainly trains relatively small CV models, which require only data parallelism and ring AllReduce algorithms.

Large models such as transformer MMOE [4] do not need to be trained, and complex cross-NPU AlltoAll traffic does not need to be processed.

Communication traffic characteristics and bandwidth requirements vary depending on computing scenarios. The communication of AI applications is mainly based on collective operations, and the main traffic types are AllReduce, AlltoAll, and AllGather. The majority of such traffic is large packets, the communication rules are predictable, and high bandwidth is required. In AI training scenarios, the high bandwidth, low latency, and collision-free features of optical signals can be fully utilized to accurately control each flow and implement contentionless communication. Exploration is growing into the application of optical communication technologies in AI training. For example, SIP-ML [6] of the Massachusetts Institute of Technology (MIT) has designed a degree-aware model parallelism policy, which uses specific optical components (OXCs and microrings) to improve training performance by 1.3 to 9.1 times. FleX-LION [31] studied by the University of California uses arrayed waveguide grating router (AWGR), microring resonator (MRR), and Mach-Zehnder switch (MZS) components to build an intra-node interconnection system, reducing the execution time to less than 1/5 of the original time. KDDI Research, Inc. in Japan proposed the pattern-aware scheduling (PAS) platform and fast convergence (FastConv) [32] to describe the control components and key policies based on the reconfigurable optical networking topology in AI scenarios from the perspective of system control. To better improve computing efficiency, network topologies, task scheduling, and communication algorithms need to be closely combined in the future. This will allow efficient AI clusters to be built by introducing the reconfigurable capabilities of optical networking [33, 34].

Compared with AI applications, HPC applications have a large amount of near-neighbor communication traffic, and the traffic is distributed diagonally. In addition, traffic distribution varies greatly in different HPC applications. For example, there are many communication types at a given moment in the early "seven dwarfs" traffic [35]. In addition to collective communication in AI applications, there are many point-to-point communication types (e.g., Send, Recv, and Barrier). Such traffic characteristics make it difficult to accurately control the direction of each flow. Furthermore, the time for computing and communication in HPC applications is irregular and out-of-order. According to the data captured in the Tianhe-2 cluster [36], the proportion of computing and communication overheads varies greatly in HPC applications. Therefore, a set of optical control policies is not suitable for different HPC applications. And because HPC scenarios have high requirements on communication latency, exploration into the use of optics in HPC scenarios focuses on building the shortest path based on optical reconfigurability and reducing detours to minimize communication latency. For example, TAGO [37] published by Columbia University and Lawrence Livermore National Laboratory (LLNL), bandwidth steering technology [38], and Flexfly [39] published by Sandia National Laboratories (SNL) have explored how optical cross-connect components are adjusted based on application traffic characteristics to provide low-latency paths between process groups with dense communication in the symmetric topology (FatTree) and asymmetric topology (Dragonfly, Dragonfly+, and Flexfly) networking environments. To achieve efficient optical interconnection communication in HPC scenarios, the following core challenges must be addressed:

1 The traffic distribution of HPC applications and bandwidth requirements of each link need to be quickly identified to promptly adjust communication paths.

2 After the topology changes, the route configuration policy in a new topology needs to be determined.

3 After the optical path is switched, data loss may occur, causing data retransmission and degrading system performance.

Data center and cloud, two mainstream scenarios, have increasingly complex application changes and more urgent requirements for flexibility. In these scenarios, there are mixed applications (both high-bandwidth applications and low-latency applications) and mixed traffic (both elephant flows and mice flows). Traffic bursts and uneven distribution are obvious [40]. In addition, these scenarios need to support features such as multi-tenant isolation and virtualization, making it more difficult to adapt to the optical interconnection structure. Currently, applications in these scenarios focus on the optimization of temporary high-bandwidth traffic and use optical paths to construct hard pipes in order to achieve high-bandwidth transmission. The main optimization policy is to identify tidal traffic and perform certain switching policies at the core switching layer to improve the overall network communication capability. Such applications include Sirius [7] of Microsoft, Gemin [41] of Google, and Helios [42] of the University of California. The core challenge of this scenario is how to properly allocate network bandwidth through the combination of traffic engineering and topology.

In addition to the preceding scenarios, an increasing amount of research introduces optical interconnection into a resource pooling scenario, for example, device resource pooling and memory resource pooling [43]. There are also studies that apply optical components to future chiplets [44]. Fully leveraging the advantages of optical components to maximize the value of optical interconnection in latency, power consumption, and bandwidth is a direction worthy of long-term exploration.

Summary

Given the trend of increasing requirements for interconnection bandwidth and switching capacity as well as enlarged cluster scale, applying optical networking to supplement for electrical networking creates a wide variety of performance, power consumption, and cost benefits. Its exploration is of great potential and economic value. This is a systematic project that requires joint innovation at all levels, including protocol, chip, and middleware levels, to achieve E2E breakthroughs. Currently, the feasibility of hybrid optical-electrical networks has been partially verified through early researches in the AI field. To meet more scenario requirements in the future, optical application solutions need to be continuously researched, developed, and finally implemented in the industry.

References

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, et al. BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[2] Alec Radford, Jeffrey Wu, Rewon Child, et al. Language models are unsupervised multitask learners. OpenAI blog, 2019, 1(8): 9.

[3] Tom Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In Proceedings of the Advances in Neural Information Processing Systems 33 (NeurIPS), 2020, 1877–1901.

[4] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to trillion parameter models with simple and efficient sparsity. arXiv preprint arXiv:2101.03961, 2021.

[5] Wei Zeng, Xiaozhe Ren, Teng Su, et al. Pangu-α: Large-scale autoregressive pretrained Chinese language models with auto-parallel computation. arXiv preprint arXiv:2104.12369, 2021.

[6] Mehrdad Khani, Manya Ghobadi, Mohammad Alizadeh, et al. SiP-ML: High-bandwidth optical network interconnects for machine learning training. In Proceedings of the ACM SIGCOMM 2021 Conference (SIGCOMM), 2021, 657–675.

[7] Hitesh Ballani, Paolo Costa, Raphael Behrendt, et al. Sirius: A flat datacenter network with nanosecond optical switching. In Proceedings of the ACM SIGCOMM 2020 conference (SIGCOMM), 2020, 782–797.

[8] Cyriel Minkenberg, Rajagopal Krishnaswamy, Aaron Zilkie, et al. Co-packaged datacenter optics: Opportunities and challenges. IET Optoelectronics, 2021, 15(2): 77–91.

[9] Brandon Buscaino, Brian D. Taylor, and Joseph M. Kahn. Multi-Tb/s-per-Fiber coherent co-packaged optical interfaces for data center switches. Journal of Lightwave Technology, 2019, 37(13): 3401–3412.

[10] CE Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. In RFC 2992

[11] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In NSDI 2010

[12] R. Ryf, J. Kim, J.P. Hickey, A. Gnauck, et al. 1296-port MEMS transparent optical crossconnect with 2.07 petabit/s switch capacity. In Proceedings of the Optical Fiber Communication Conference and Exhibit. Technical Digest Post Conference Edition, 2001, 28–28.

[13] J. Kim, C. J. Nuzman, B. Kumar, et al. 1100 x 1100 port MEMS-based optical crossconnect with 4-dB maximum loss. IEEE Photonics Technology Letters, 2003, 15(11): 1537–1539.

[14] David T. Neilson, R. Frahm, Paul Kolodner, et al. 256 × 256 port optical cross-connect Subsystem. Journal of Lightwave Technology, 2004, 22(6): 1499–1509.

[15] Keijiro Suzuki, Ryotaro Konoike, Junichi Hasegawa, et al. Low-insertion-loss and power-efficient 32× 32 silicon photonics switch with extremely high-Δ silica PLC connector. Journal of Lightwave Technology, 2019, 37(1): 116–122.

[16] Chang Chang, Ting Li, Yulin Wu, et al. Fast-response, energy-efficient thermo-optic silicon phase shifter based on non-Hermitian engineering. In Proceedings of the 2022 Optical Fiber Communications Conference and Exhibition (OFC), 2022, 1–3.

[17] Lei Qiao, Weijie Tang, and Tao Chu. 32 × 32 silicon electro-optic switch with built-in monitors and balanced-status units. Scientific Reports, 2017, 7 (42306): 1–7.

[18] Mingwei Jin, Jia-Yang Chen, Yong Meng Sua, et al. High-extinction electro-optic modulation on lithium niobate thin film. Optics Letters, 2019, 44(5): 1265–1268

[19] Shengqian Gao, Mengyue Xu, Mingbo He, et al. Fast polarization-insensitive optical switch based on hybrid silicon and lithium niobate platform. IEEE Photonics Technology Letters, 2019, 31(22): 1838–1841.

[20] Amirmahdi Honardoost, Johannes Henriksson, Kyungmok Kwon, et al. Low-loss wafer-bonded silicon photonic MEMS switches. In Proceedings of the 2022 Optical Fiber Communications Conference and Exhibition (OFC), 2022, 1–3.

[21] Honglin Zhang, "Tengxun daguimo fenbushi jiqi xuexi xitong wuliang shi ruhe jinxing jishu xuanxing de?" [How does Tencent select technologies for its large-scale distributed machine learning system (Wuliang)?], in InfoQ, 2018.

[22] Chris Ying, Sameer Kumar, Dehao Chen, et al. Image classification at supercomputer scale. arXiv preprint arXiv:1811.06992, 2018.

[23] Kun Tan, Heng Liao, Lijun Li, et al. UB: Unified and Scalable Memory-Semantic Interconnection for Next-Generation Computing Systems. Communications of Huawei Research, 2021, 1(7): 118–127.

[24] Tirthak Patel, Zhengchun Liu, Raj Kettimuthu, et al. Job characteristics on large-scale systems: Long-term analysis, quantification, and implications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2020, 1–17.

[25] Staci A. Smith, Clara E. Cromey, David K. Lowenthal, et al. Mitigating Inter-Job Interference Using Adaptive Flow-Aware Routing. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2018, 346–360.

[26] Nathan Binkert, Bradford Beckmann, Gabriel Black, et al. The gem5 simulator. ACM SIGARCH computer architecture news, 2011, 39(2): 1–7.

[27] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In Proceedings of Machine Learning and Systems, 2021, 829–844.

[28] Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, et al. ASTRA-SIM: Enabling SW/HW co-design exploration for distributed DL training platforms. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2020, 81–92.

[29] Han Lin, Hong An, Mingfan Li, et al. TripletRun: A dataflow runtime simulator and its performance model. In Proceedings of the IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, 1509–1517.

[30] Zhihao Jia, Sina Lin, Charles R. Qi, et al. Exploring hidden dimensions in parallelizing convolutional neural networks. In Proceedings of the 35th International Conference on Machine Learning (ICML), 2018, 2279-2288.

[31] Marjan Fariborz, Xian Xiao, Pouya Fotouhi, et al. Silicon photonic flex-LIONS for reconfigurable multi-GPU systems. Journal of Lightwave Technology, 2021, 39(4): 1212-1220.

[32] Cen Wang, Noboru Yoshikane, Filippos Balasis, et al. Acceleration and efficiency warranty for distributed machine learning jobs over data center network with optical circuit switching. In Proceedings of the 2021 Optical Fiber Communications Conference and Exhibition (OFC), 2021, 1-3.

[33] M. Ghobadi. Emerging optical interconnects for AI systems. In Proceedings of the 2022 Optical Fiber Communications Conference and Exhibition (OFC), 2022, 1-3.

[34] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, et al. TopoOpt: Optimizing the network topology for distributed DNN training. arXiv preprint arXiv:2202.00433, 2022.

[35] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, 2006.

[36] Wenhao Zhou, Juan Chen, Zhiyuan Wang, et al. Time-dimension communication characterization of representative scientific applications on Tianhe-2. In Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), and 2015 IEEE 12th International

Conference on Embedded Software and Systems (ICESS), 2015, 423-429.

[37] Min Yee The, Yu-Han Hung, George Michelogiannakis, et al. TAGO: Rethinking routing design in high performance reconfigurable networks. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2020, 1-16.

[38] George Michelogiannakis, Yiwen Shen, Min Yee Teh, et al. Bandwidth steering in HPC using silicon nanophotonics. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2019, 1–25.

[39] Ke Wen, Payman Samadi, Sébastien Rumley, et al. Flexfly: Enabling a reconfigurable dragonfly through silicon photonics. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2016, 166–177.

[40] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM conference on Internet Measurement (IMC), 2010, 267–280.

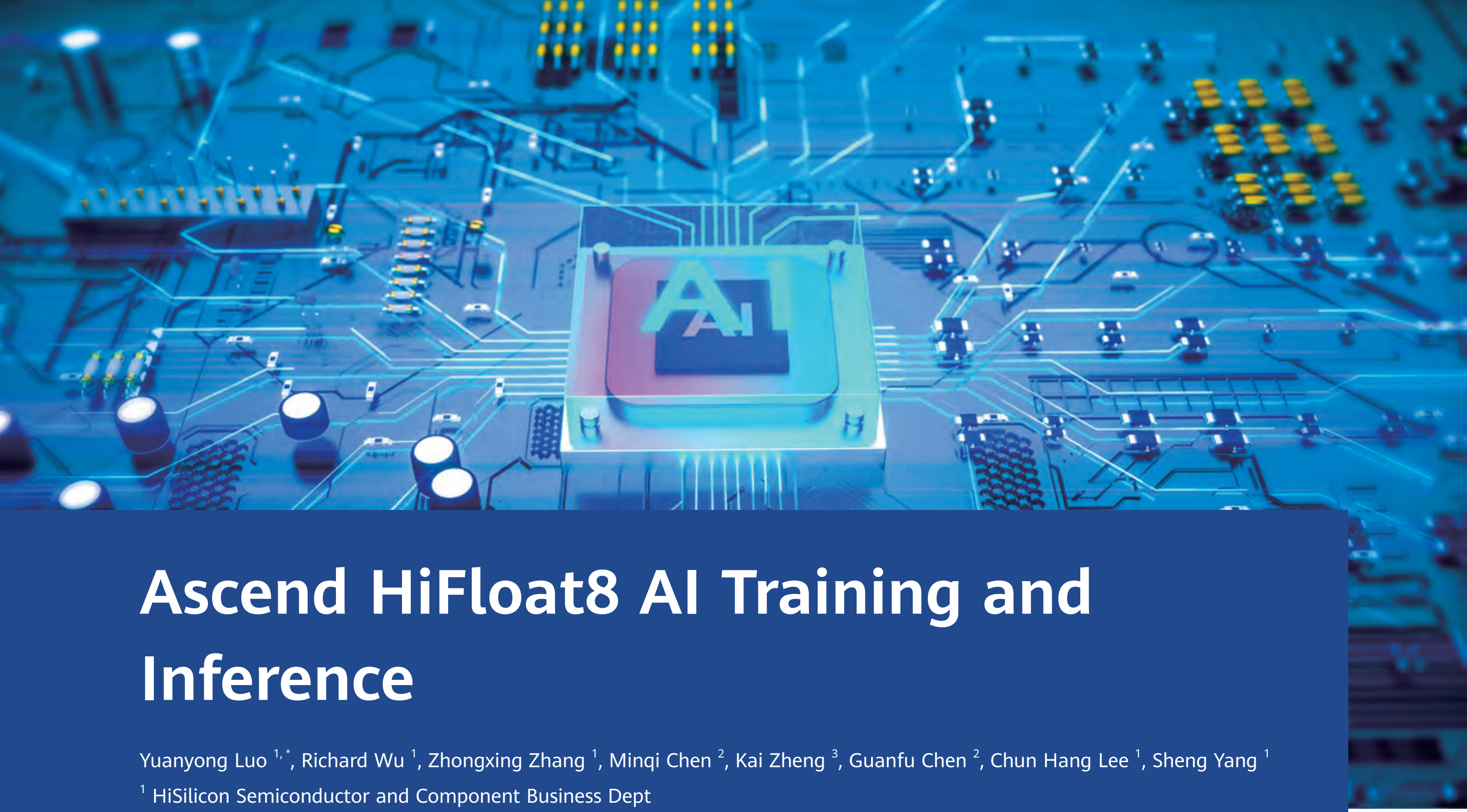
[41] Mingyang Zhang, Jianan Zhang, Rui Wang, et al. Gemini: Practical reconfigurable datacenter networks with topology and traffic engineering. arXiv preprint arXiv:2110.08374, 2021.

[42] Aathan Farrington, George Porter, Sivasankar Radhakrishnan, et al. Helios: a hybrid electrical/optical switch architecture for modular data centers. In Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM), 2010, 339–350.

[43] Arastu Sharma, Qixiang Cheng, Nikolaos Bamiedakis, et al. Bandwidth reconfigurable optical switching architecture for CPU-GPU computing systems with shared memory. In Proceedings of the Optical Fiber Communication Conference (OFC), 2022, W2A.19.

[44] Arastu Sharma, Nikolaos Bamiedakis, Fotini Karinou, et al. Multi-Chiplet system architecture with shared uniform access memory based on board-level optical interconnects. In Proceedings of the 2021 Optical Fiber Communications Conference and Exhibition (OFC), 2021, 1–3.

[45] NVIDIA DGX SuperPOD. https://www.nvidia.com/en-us/data-center/dgx-superpod/



Ascend HiFloat8 AI Training and Inference

Yuanyong Luo [1,*], Richard Wu [1], Zhongxing Zhang [1], Minqi Chen [2], Kai Zheng [3], Guanfu Chen [2], Chun Hang Lee [1], Sheng Yang [1]

[1] HiSilicon Semiconductor and Component Business Dept

[2] Central Media Technology Institute

[3] Central Hardware Engineering Institute

Abstract

This paper proposes the innovative 8-bit floating-point format HiFloat8 (abbreviated as HiF8), HiF8-based AI training solution, and HiF8-based AI inference solution.

As a single data format, HiF8 can be simultaneously used in the forward pass and backward pass of AI training. With HiF8, both network statistics analysis and switching between different Float8 data formats are not required during training. HiF8 features tapered precision and balances the precision and dynamic range, eliminating the need for dynamic matching between data and formats during training, which causes redundant data scaling or exponent sliding window operations. The HiF8 training process is the same as the existing FP16 mixed precision training process. Therefore, HiF8 is intuitive and easy to use, and users are not required to know its details. The verification experiments of large-scale training show that compared with the accuracy of FP16 mixed precision training, the accuracy of HiF8 training on convolutional neural networks (CNNs) decreases by 0.16% (0.12% if FP16 is allowed to be used for training at the input layer) on average and the accuracy of HiF8 training on Transformer networks increases by 0.12% on average.

In terms of inference, this paper proposes a target metric loss–oriented hierarchical calibration solution that is applicable to HiF8. The verification experiments of large-scale inference show that:

- Transformer networks can be used for inference after the data type is directly converted to HiF8.
- Most CNNs can be used for inference after direct data type conversion to HiF8 + calibration.
- A few CNNs that are difficult to calibrate can be used for inference if some layers use FP16 while others use HiF8 for computing.

This paper also evaluates the overheads and benefits of HiF8 based on the electronic system level (ESL) models of Ascend *XXX*1 and *XXX*2. When Ascend *XXX*1 serves as the baseline, if the computing power of HiF8 is twice that of FP16, the area of AI Core increases by approximately 4.5%, and the training performance of ResNet50 and BERT can be improved by 26% and 61%, respectively. If Ascend *XXX*2 serves as the baseline, the training performance of ResNet50 and BERT can be improved by 31% and 67%, respectively.

* Corresponding author

1 Development of AI Data Formats

In 2012, AlexNet [1] won the championship of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), kicking off an explosion of deep convolutional neural networks (DCNNs). In 2017, the Transformer [2] architecture was proposed, which ushered in the era of large-scale deep neural network (DNN) models with hundreds of millions of parameters. Since then, the parameter scale of large-scale Transformer models has been increasing exponentially by 240 times every two years. In addition to the silicon-based technologies developed according to Moore's Law [3] and architecture innovation, AI data formats are also continuously developing to meet the computing power requirements of AI. The development of AI data formats can be divided into the following four phases based on the time for commercial use:

Phase 1 (1956–2006): FP64. The concept of AI was first proposed in 1956. The core idea was to enable machines to have basic logical inference capabilities. In 1970, the backpropagation algorithm was proposed [4]. Since 1980, the knowledge base system and knowledge engineering have become the main directions of AI research. In 1982, the Hopfield network was proposed [5]. In 1997, Deep Blue beat the world chess champion. From 1962 to 2006, the theoretical basis of deep learning, including theories related to CNNs and long short-term memory (LSTM), was established and developed but was not mature. During this period, CPUs were the major AI hardware, and the mainstream data format was the 64-bit double-precision floating-point format FP64 defined in IEEE Standard for Floating-Point Arithmetic (IEEE 754) in 1985 [6]. Before this international standard was established, the 60-bit floating-point format (predecessor of FP64) applied to CDC 6000 was in use.

Phase 2 (2006–now): FP32. In 2006, Hinton et al. successfully trained a multilayer feedforward neural network, indicating that the deep learning theory had become mature [7]. The CPU-based FP64 data format gradually became incapable of meeting AI's increasingly higher computing power requirements being driven by deep learning. Researchers were in urgent need of greater computing power to accelerate general matrix-vector multiplication (GEMV) and general matrix-matrix multiplication (GEMM). In 2006, FP32 was used for the first time to train CNNs on GPUs and achieved a four-fold performance improvement compared with FP64 training on CPUs [8]. The bit width of FP32 is only half that of FP64. Therefore, a chip that uses the FP32 format can store more data and integrate more multiply-accumulate (MAC) units than one that uses FP64. The single instruction multiple data (SIMD) computing mode enables GPUs to have a significantly higher degree of computing parallelism than CPUs. AlexNet [1] which ran on two GPUs won the championship of ILSVRC 2012, making FP32 the mainstream data format for deep learning training from 2012 to 2017.

Phase 3 (2017–now): Float16 mixed precision. In this phase, the scale of deep learning networks was increasing exponentially, whereas Moore's Law was slowing down, widening the gap between AI computing requirements and the computing density of hardware. In addition, the memory wall problem became increasingly severe. The academia started to explore AI data formats with smaller bit widths than FP32. The explorations included binary quantization [9], ternary quantization [10], and 4- to 6-bit quantization of activation values or weights [11], dynamic quantization of weights, activation values, and gradients [12], and 16-bit fixed-point training of neural networks [13]. These radical explorations either caused a sharp decrease in the training accuracy or were verified to be applicable to small datasets only. In 2015, Google proposed a novel Float16 data format in its TensorFlow white paper. This format was equivalent to FP32 without the last 16 bits and was used as the input data format of multiplication, whereas FP32 was used for accumulation in the GEMM [14, 15]. This format was later named Brain Floating Point 16 (BF16) [16] and deployed on Tensor Processing Unit (TPU) V2 and V3. In 2017, Google's AlphaGo deployed based on TPU V2 defeated the world's number one Go player Ke Jie in all three games, causing a global sensation. Unlike Google, NVIDIA and Huawei adopt the IEEE 754 half-precision floating-point format [17]. NVIDIA released the V100 GPU [18] in 2017 and Huawei released the Ascend NPU [19] in 2019. Both the GPU and NPU support the FP16 and FP32 mixed precision training strategies. Loss scaling needs to be introduced into the backward pass of FP16 training to prevent a large number of zero-valued gradients because the dynamic range of FP16 is small [20]. Since 2017, the Float16 mixed precision training solution, mainly consisting of BF16 and FP16, has become the main choice for deep learning training. However, FP32 is still used for some networks requiring high precision. Over the same period, some other customized AI data formats were also put forward to compete with BF16 and FP16. In 2019, IBM proposed the DLF16 data format [21] consisting of 1 sign bit, 6 exponent bits, and 9 mantissa bits. The training strategy based on DLF16 mixed precision has not been fully verified on a large number of neural networks. The A100 GPU released by NVIDIA in 2020 supports mixed precision training based on TF32 — a 19-bit AI data format. This format is equivalent to FP32 without the last 13 bits and applies to neural networks that require a higher precision than that of Float16 [22]. The 19-bit width of TF32 is not conducive to data storage or transfer. In addition to the preceding floating-point formats with a fixed bit width for each field, the floating-point format posit [24] was invented by John Gustafson in 2017 based on Unum [23]. Posit has a mantissa bit width that features tapered precision within the exponent range. However, the 16-bit posit mixed precision training was carried out only on a small number of small-scale networks or datasets [25]. In general, BF16 and FP16 have achieved decent results in Float16 mixed precision training. Most existing DNNs are based on the BF16 + FP16 Float16 mixed precision training.

Phase 4 (2021–now): Float8 mixed precision. Google ushered in the era of Float16 mixed precision training, whereas IBM led the exploration of Float8 mixed precision training. In 2018, IBM radically cut off the last 8 bits of FP16 to obtain the FP8 data format with 1 sign bit, 5 exponent bits, and 2 mantissa bits (1-5-2) [26], and proposed the GEMM computation method in which FP8 is used for multiplication and FP16 (1-6-9) is used for accumulation. The FP8 training solution achieved good results in ResNet50 training when stochastic rounding (SR) and backward loss scaling were supported. However, on MobileNetV2 and Transformer networks, the accuracy of FP8 training decreased dramatically. On this basis, IBM proposed the 8-bit hybrid FP8 (HFP8) training solution in 2019 [27]. For GEMM, HFP8 uses FP8 (1-4-3) for multiplication in the forward pass of training, FP8 (1-5-2) for multiplication in the backward pass of training, and FP16 (1-6-9) for accumulation. HFP8 has achieved desired training results on some CNN, LSTM, and Transformer networks. The HFP8 training solution uses different data formats in the forward pass and backward pass of training, and therefore it is also called FP9 in GEMM hardware implementation [28]. To avoid the trouble of selecting one of the two FP8 data formats during training, the FP8 solution supporting shared exponent bias (SEB) was proposed [29, 30]. Based on an FP8 format with a large mantissa bit width, this solution adds a proper bias to the exponent of a piece of data through GEMM-level data exponent sharing, so as to adjust the dynamic range of the data format for data matching. In August 2021, Tesla took the lead in deploying a Float8 mixed precision training solution called configurable floating point 8 (CFP8) on its new product Dojo [31]. CFP8 uses the FP8 (1-4-3) and FP8 (1-5-2) formats of HFP8, each of which is configured with a 6-bit SEB. At the end of November 2021, Amazon deployed configurable Float8 (cFP8), another Float8 mixed precision training solution on its new product EC2 Trn1 [32]. The cFP8 solution involves three FP8 data formats: FP8 (1-5-2), FP8 (1-4-3), and FP8 (1-3-4). However, it does not support the SEB. In addition, the cFP8 mixed precision training solution supports SR. In March 2022, NVIDIA also deployed the Float8 mixed precision training solution on its new product H100 GPU [33]. NVIDIA adopts the FP8 (1-4-3) and FP8 (1-5-2) formats of HFP8 and introduces an additional hardware module called the Transformer Engine to accelerate the training of Transformer networks. This module analyzes statistics, determines whether to use 1-4-3, 1-5-2, or FP16 as the format of the GEMM input data at the current layer of the Transformer network, and implements Float8 + Float16 dynamic mixed precision training. In addition, after the Transformer Engine determines whether to use FP8 (1-4-3) or FP8 (1-5-2) at the current layer, a scaling factor needs to be computed to scale data into the dynamic range of FP8. NVIDIA claimed that the combination of FP8 and Transformer Engine could reduce the time required for training Transformer models from weeks to days. Table 1 compares various Float8 solutions in the industry. On the whole, the commercial Float8 solutions of major vendors have certain similarities with IBM's HFP8.

Table 1 Comparison of Float8 solutions in the industry

Item	IBM HFP8	Tesla CFP8	Amazon cFP8	NVIDIA FP8
Format	1-4-3/1-5-2	1-4-3/1-5-2	1-3-4/1-4-3/1-5-2	1-4-3/1-5-2
SEB	-	6 bits	-	-
Scaling	N	N	Y*	Y
Statistics	-	Software*	Software*	Transformer Engine
SR	N	Undisclosed	Y	Undisclosed
NaN/Inf Support	N	N	Undisclosed	Undisclosed
*: Conjecture				

On the one hand, with the evolution of deep learning algorithms, neural networks have an increasingly higher tolerance for computational errors. On some neural networks, the accuracy even slightly improves after training due to a lower computing precision. The reason is that increasing the computational error is equivalent to introducing noise, avoiding overfitting to a certain extent, and enhancing the generalization capability of such neural networks [34]. On the other hand, the exponential growth in the network scale has caused severe problems such as insufficient computing power and memory walls [35]. If the bit widths of data formats can be greatly reduced, the computing power can be significantly improved or the memory wall problem can be mitigated. Based on the preceding two aspects, AI data formats have been developing toward smaller bit widths in recent decades, from 64-bit width to the 32-bit width, and then to 16-bit mixed precision. The era for commercialization of 8-bit mixed precision data formats has arrived.

In the following sections, we will break down, analyze, and discuss existing Float8 mixed precision solutions in depth, and introduce the Huawei proprietary HiF8 data format and Ascend HiF8 mixed precision training and inference solutions.

2 Rethinking Float8 and AI

As described above, training accuracy and computing precision of deep learning networks are not positively correlated. The accuracy of FP32 training is slightly higher than that of Float16 mixed precision training on some networks, whereas the result may be the opposite on other networks, depending on the network structures. This is because the error or noise caused by a decrease in computing precision is double-edged. If the error or noise is within a specific range, the network generalization capability is enhanced. If the error or noise is beyond this range, the network generalization capability is weakened because the computational error is excessively large or small. An excessively large computational error will lead to a significant decrease in training accuracy. During the transition from FP32 to Float16 mixed precision, Float16 itself is within the inflection point range with relatively good computing precision and network generalization capability. Therefore, both BF16 which favors dynamic range, and FP16 which favors computing precision + backward loss scaling have achieved commercial success. In cases where simple floating-point data formats with fixed field bit widths can achieve good results, the industry does not consider other complex data formats with high overheads.

However, during the transition from Float16 to Float8, the computing precision decreases to the critical point and the dynamic range also reaches the critical point, which requires deliberation. In this case, the computing precision and dynamic range need to be balanced. Based on a series of explorations originating from IBM's Float8, the current implementation in the industry, and a large number of experiments we have conducted, the following conclusions can be drawn:

- A single Float8 data format with a fixed field bit width either has low computing precision or a small dynamic range (the exponent window is too narrow) and therefore cannot meet the network training requirements. The forward pass of training requires high computing precision, whereas the backward pass of training requires a large dynamic range.
- When multiple Float8 data formats with fixed field bit widths are dynamically mixed, the data and format still need to be matched if the selected Float8 data formats favor high computing precision and have small dynamic ranges. For example, the forward pass of training requires high computing precision. However, the training accuracy decreases dramatically if the dynamic range does not properly match the statistical characteristics of the data.
- There are two methods for matching data and formats. One is to scale data at the GEMM level so as to match the exponent window of Float8. The other is to set a GEMM-level SEB, that is, to move the exponent window of Float8 to match the data. Amazon and NVIDIA have chosen the former, whereas Tesla has chosen the latter.

The Float8 mixed precision solution based on a fixed field bit width has the following problems:

- In terms of hardware, extra overheads are introduced. For example, multiple Float8 data formats need to be identified by hardware. GEMM-level matching between formats and data requires the SEB or Transformer Engine.
- In terms of software, additional algorithms are required to select formats and match the data and formats.
- In terms of users, the fine-grained complexity (i.e., temporal locality and spatial locality) at the GEMM level can be perceived by users and is therefore not user-friendly.

In the phase when Float16 was the research focus, the 16-

bit posit is a decent exploration result among floating-point formats with non-fixed field bit widths. Posit features tapered precision. Its mantissa bit width is relatively large within the data range in which the absolute value is close to 1, but gradually decreases to 0 as the absolute value is farther away from 1. However, posit encoding and decoding are too complex and require a relatively large multiplier bit width and high hardware overhead. Compared with BF16 and FP16 which can achieve fair training effects, posit16 slightly improves the training accuracy, but it has an obvious drawback of high hardware overhead. In the competition with Float8, a single posit8 data format cannot achieve a proper tradeoff between the dynamic range and computing precision. The applicable networks and datasets are still limited even if the operation of matching formats and data is performed [36, 37]. In addition to posit, another floating-point format with non-fixed field bit widths — EFloat — was proposed by IBM [38]. EFloat collects information entropy statistics and then performs Huffman coding on the exponent, enabling the mantissa bit width to change dynamically. The complexity granularity of EFloat is finer than that of the preceding floating-point solutions, and therefore more details are not provided in this paper.

Inspired by the floating-point solutions with non-fixed field bit widths, we hold that a feasible approach to solving the problems of preceding Float8 solutions with fixed field bit widths is to explore the bit width allocation rules of the exponent and mantissa in a single Float8 data format. During AI training, data distribution presents a centralized characteristic similar to that of Gaussian distribution. Therefore, posit with tapered precision is a perfect choice. However, posit encoding and decoding are too complex, and a single posit8 data format has either a small dynamic range or low big data computing precision. Therefore, a new data format needs to be designed. Considering that FP16 mixed precision has achieved business success and automatically supports loss scaling in the backward pass of training, we hope to design a new data format that has the same 5-bit equivalent exponent as FP16 while maintaining tapered precision. Such a data format can maintain the same mode as FP16 mixed precision during AI chip design, deployment, and use. Therefore, it is user-friendly and its software and hardware deployment is simple.

3 HiFloat8

This section describes the definition of the novel 8-bit floating-point data format HiFloat8 (abbreviated as HiF8) and its support for special values. We then elaborate on the rounding support of HiF8, which is a focus of AI training and inference.

3.1 HiF8 Data Format

HiFloat is a new general-purpose floating-point encoding and decoding mode and data expression mode. The bit width, data range, and significant bits of HiFloat can be scaled based on scenario requirements. HiFloat applies to various fields such as general-purpose computing, high-performance computing (HPC), and AI. This paper focuses on the 8-bit floating-point instance HiF8 of HiFloat in AI low-precision training and inference scenarios. HiF8 defines an additional dot field on the basis of IEEE 754 [17]. Therefore, HiF8 consists of the four fields listed in Table 2: a sign field, a dot field, an exponent field, and a mantissa field.

Table 2 Definition and bit width of each field in HiF8

Field	Sign (S)	Dot (D): {Values}	Exponent (E)	Mantissa (M)
Width	1	2: {2, 3, 4}	D	5 – D
	1	3: {0, 1}	D	4 – D

The following describes each field in detail:

- Sign field: 1 bit, indicating the positive and negative signs of HiF8 data. By default, 1 indicates the negative sign and 0 indicates the positive sign.
- Dot field: 2 or 3 bits used to code the five D values (0 to 4). D explicitly indicates the number of bits occupied by the exponent field, and implies the number of bits occupied by the mantissa field. The dot field is coded by using unconventional prefix codes, that is, the 3-bit width is used for coding small values 0 and 1, whereas the 2-bit width is used for coding large values 2, 3, and 4. Table 3 describes the default mapping.
- Exponent field: D bits, where D is equal to the coded value of the dot field and $D \in \{0, 1, 2, 3, 4\}$. The exponent field uses signed-magnitude codes to represent values, but the most significant bit (MSB) of the magnitude is fixed to 1. The 1-bit sign of the exponent is marked as Se. By default, the exponent sign is positive when Se is 0, and negative when Se is 1. In binary mode,

Table 3 HiF8 dot field coding

Dot: Unconventional Prefix Code		
Width	**Coding**	**Value**
2	11	4
	10	3
	01	2
3	00, 1	1
	00, 0	0

the binary signed-magnitude code of the exponent can be completely expressed as Ei:

$$Ei = \{Se, Mag[1{:}end]\} = \{Se, 1'b1, Mag[2{:}end]\}$$

Since the MSB of the magnitude is fixed to 1, this bit can be hidden and Ei can be expressed as Es during storage:

$$Es = \{Se, Mag[2{:}end]\}$$

The number of bits in Es equals the value of D. When D is 0, Es does not occupy any bit width, indicating that the exponent value equals 0. Table 4 describes the exponent coding details of HiF8.

In Table 4, E is a decimal numerical representation of the exponent. The exponent window of HiF8 is [–15, +15], and therefore there are 31 exponent values in total.

- Mantissa field: (5–D) bits, where D is equal to the coded value of the dot field and $D \in \{0, 1, 2, 3, 4\}$. The mantissa field stores only M, that is, the value in fractional bits. The actual value is 1.M or 1 + M. The MSB of the significand is fixed to 1, hidden, and not stored. The mantissa field has four fractional bits when the exponent center of HiF8 equals 0 (i.e., E equals 0), as described in Table 4. When the absolute value of E gradually increases, the bit width of the mantissa field gradually decreases to 1.

A HiF8 floating-point number can be expressed by the following formula:

$$X = (-1)^S \times 2^E \times (1 + M)$$

S, E, and M indicate the values of the sign, exponent, and mantissa fields of HiF8, respectively.

In addition, to cope with various special cases in HiF8 numerical computation and facilitate debugging during AI training and inference, HiF8 defines four special values: Zero, not a number (NaN), and positive/negative infinity (±Inf). These special values are described as follows:

- If S = 0, D = 4, Es = 4'b1111 (E = –15), and M = 1'b0, then X = Zero.
- If S = 1, D = 4, Es = 4'b1111 (E = –15), and M = 1'b0, then X = NaN.
- If D = 4, Es = 4'b0111 (E = 15), and M = 1'b1, then X = ±Inf.

Figure 1 shows the distribution of significant bits (hidden 1-bit integer + mantissa: 1.M) of HiF8 and HFP8 in the exponent window.

The maximum and minimum positive numbers that can be expressed by HiF8, that is, the dynamic range of the absolute values, are as follows:

$$X_{max\_pos} = 2^{15} = 32768 \ (Including\ Inf)$$

$$X_{min\_pos} = 2^{-15} \times 1.5 = 0.000045776367 \ (Including\ NaN\ \&\ Zero)$$

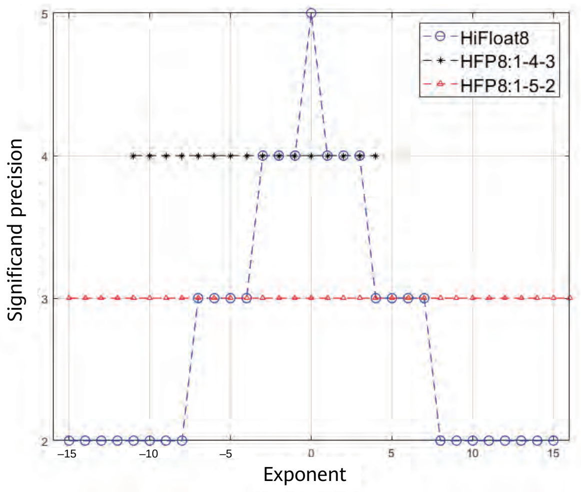


Figure 1 Significand precision on exponent of HiF8 and HFP8

Table 4 HiF8 code-value mapping details

Value of D	0	1	2	3	4
Es	None	Se	Se, Mag[2]	Se, Mag[2:3]	Se, Mag[2:4]
Ei	0	Se, 1	Se, 1, Mag[2]	Se, 1, Mag[2:3]	Se, 1, Mag[2:4]
E	0	±1	±[2, 3]	±[4, 7]	±[8, 15]
Width of M	4	3	3	2	1

In summary, as a single Float8 data format, HiF8 balances the dynamic range and computing precision. It has the following four major characteristics:

- Tapered precision: When the HiF8 exponent is extended from the center to both sides, the number of significant bits gradually changes from 5 to 2. HiF8 uses unconventional prefix codes for numerical coding of the dot field, preventing problems caused by using conventional prefix codes, including the precision jump problem and the problem that the number of significant bits of large values is too small.
- 5-bit equivalent exponent: Same as FP16, HiF8 has a 5-bit exponent equivalently. This helps HiF8 maintain the same neural network training process as FP16 mixed precision. Therefore, HiF8 features easy software and hardware deployment and is user-friendly.
- No redundant information: HiF8 uses the signed-magnitude representation for coding and hides the magnitude MSB that is fixed to 1 during exponent storage, ensuring that the exponent values under different D values are unique. HiF8 hides the integer bit that is fixed to 1 during mantissa storage.
- Support for various special values: HiF8 defines four key special values. These special values play a significant role in debugging during AI training and inference. However, commercial solutions usually use multiple Float8 data formats, making it difficult to define and support special values NaN and ±Inf in a unified manner.

3.2 HiF8 Rounding Support

In Float8 mixed precision training and inference, high-precision floating-point format such as FP32 needs to be converted into low-precision format Float8 and then input to the GEMM, during which rounding is involved. As the precision of Float8 is low, the rounding method is extremely sensitive to the convergence and accuracy of neural network training. After theoretical analysis and a large number of experiments, we conclude that HiF8 supports two rounding methods during format conversion: round half to away (TA) and SR. In addition, to meet the requirements of certain AI algorithms, HiF8 also provides two options: saturation to boundary values upon overflow and NaN saturation to zero. The following describes the TA and SR rounding solutions used during the conversion from FP32 to HiF8:

- Round half TA

To reduce the computational error, HiF8 preferentially uses the rounding solution with an error of 0.5 unit of least precision (ulp) only, that is, round half. Directed rounding solutions with an error of 1 ulp, such as round up, round down, and round to zero, are not considered because they cause excessively large computational errors and severe mean shift during the training of neural networks, making it difficult for the training to converge. Round half is classified into round half to even (TE) and round half TA [17]. In TE rounding, if the MSB of the discarded bits is 1 and other discarded bits are 0, the last bit of the reserved bits is checked. If the last bit is 1, the reserved bits increase by 1. If the last bit is 0, the reserved bits remain unchanged, that is, the number obtained after rounding is even. If the MSB of the discarded bits is x (0 or 1) and the other bits are not 0, x is added to the reserved bits. In most papers and commercial products, TE rounding is used by default because it maximizes the unbiasedness of round half. In contrast, TA rounding focuses only on the MSB (x) of the discarded bits and directly adds x to the reserved bits. Therefore, TA rounding is biased rounding in the special case of TE rounding (i.e., the case where the MSB of the discarded bits is 1 and other discarded bits are 0). In conclusion, TE rounding enhances the unbiasedness of round half, whereas TA rounding features easier hardware implementation.

The occurrence probability of the TE special case is extremely low during the conversion from FP32 to HiF8. For example, the probability is 2^{-20} if HiF8 needs to reserve three fractional bits. The biggest challenge of low-precision Float8 in AI training and inference is its limited data resolution capability. The analysis result shows that the data resolution capability of TA is slightly higher than that of TE during the conversion from FP32 to HiF8. In consideration of the TE special case, it is assumed that there are three 3-bit numbers with consecutive integer bits: 00.1, 01.1, and 10.1. The round half TE result of the three values is 00 and 10, whereas the round half TA result is 01, 10, and 11. Therefore, in the TE special case with an extremely low occurrence probability, TA enables a higher data resolution capability of Float8 than TE although TA is biased rounding, and the errors of TE and TA are both 0.5 ulp. The simulation experiments of HiF8 mixed precision training show that the accuracy of networks trained based on TA is slightly higher than that of networks trained based on TE because TA enables a slightly higher data resolution capability than TE. For example, the TA-based training accuracy of ResNet50 and that of MobileNet_V2 are 0.06% and 0.11% higher than the TE-based training accuracy of ResNet50 and that of MobileNet_V2, respectively.

In summary, compared with TE, TA features simpler hardware implementation and higher training accuracy. Therefore, HiF8 uses round half TA.

- SR

  Large-scale HiF8 mixed precision training experiments show that global TA rounding applies to almost all DNNs, except YoLo_V3_Tiny. When only TA rounding was used for HiF8 training on YoLo_V3_Tiny, some segments crashed. As a result, the final accuracy was 1.67% lower than the FP32 baseline. After a lot of research and many experiments, we propose a HiF8 training solution that combines TA and SR (details will be provided later) and enables the YoLo_V3_Tiny network training accuracy to be close to the baseline value. Therefore, HiF8 supports both TA rounding and SR.

  The error of SR is 1 ulp. Compared with TA rounding, SR has a significant advantage when data is processed in batches. Specifically, in SR, a uniformly distributed random number needs to be randomly generated and used as threshold T ($T \in [0, 1)$). All bits to be discarded are regarded as fractional bits and marked as F ($F \in [0, 1)$). F and T are then compared. If $F \geq T$, 1 is added to the reserved bits (K). If $F < T$, 0 is added to the reserved bits (K), that is, K remains unchanged. As threshold T is uniformly distributed, the expected value after SR is expressed as follows:

  $$(K + 1) \times F + K \times (1 - F) = K + F$$

  SR can maintain the overall mean invariance during batch data rounding to the maximum extent [39].

  Because DNNs need to generate a large number of uniformly distributed random numbers in parallel, both software and hardware implementations of SR hit a performance bottleneck [40]. To tackle the bottleneck, we come up with a simplified SR hardware implementation solution. Take the conversion from FP32 to HiF8 as an example. Assuming that the value of the most significant 14 bits (F14) of the discarded bits approximately equals F, the precision is high enough. Then, the least significant 14 bits (T14) of the discarded bits are used as the random number threshold for uniform distribution (both theoretical analysis and experiments show that the lower mantissa bits of floating-point numbers follow uniform distribution). Then, F14 and T14 are compared to complete simplified SR. A large amount of end-to-end (E2E) HiF8 training shows that the effect of this method is basically the same as that of the methods mentioned earlier.

In conclusion, the combination of TA rounding and SR can solve the convergence problem of very few special networks. In addition, the simplified SR hardware implementation solution proposed in this paper has a low overhead and does not hit any bottleneck during random number generation. Therefore, HiF8 also supports SR.

4 HiF8 AI Training

This section describes the HiF8 mixed precision training solution, including the HiF8 training process, the method for setting up a HiF8 simulation experiment platform that can be used for both training and inference, and two application scenarios of HiF8 AI training.

4.1 HiF8 AI Training Process

As mentioned above, the mantissa bit width of HiF8 gradually decreases in the process of extending the exponent from the center to both sides, that is, HiF8 features tapered precision. Based on this feature, HiF8 can balance the computing precision and dynamic range. The number of significant bits ranges from 2 to 5. The exponent bit width of HiF8 is 5, which is the same as that of FP16. The dynamic range of HiF8 is [–32768, +32768], which is quite close to that of FP16. Similar to FP16 training, HiF8 training does not require analysis of statistics at each network layer or matching between formats and data. Different from the training solution involving multiple Float8 data formats with fixed field bit widths, HiF8 is a single data format, and therefore switching between different Float8 data formats is not needed during training. To sum up, HiF8 can directly use the AI training solution of FP16 mixed precision to prevent an increase in the costs of user learning and transfer. Figure 2 shows the training process of HiF8 mixed precision.

This paper describes the HiF8 mixed precision training solution from the following four aspects:

- Forward pass of training

  For GEMM operations performed at the input and intermediate layers of the network, including Convolution, Convolution Transpose, Full Connection,

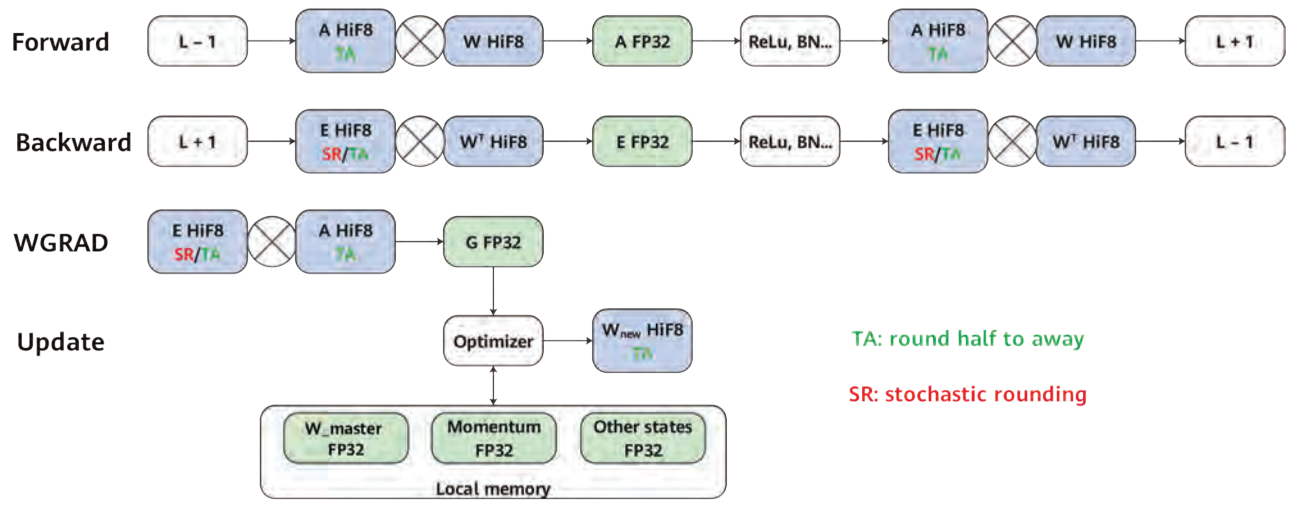


Figure 2 Training process of HiF8 mixed precision

and MatMul, the data of different formats (FP32, FP16, Int8, etc.) needs to be converted into HiF8 data and then input to the GEMM computing unit, which then outputs FP32 or FP16 data. The activation layer (ReLU, GeLU, etc.) and normalization layer (batch normalization, layer normalization, etc.) use the FP32 or FP16 data format for computing. The output layer of the neural network retains the FP16 or FP32 data format.

- Backward pass of training

  HiF8 mixed precision inherits the loss scaling operation in the backward pass of FP16 training. That is, the loss is multiplied by a proper scaling value, and then derivation is performed in a backward manner. During the derivation of the feature map and weight, the GEMM input is of the HiF8 data type (FP16 or FP32 is still used at the output layer of the neural network), and the activation layer and normalization layer use the FP32 or FP16 data type.

- Weight update process

  The gradient corresponding to the weight obtained in the backward pass is stored as FP32 data. The computation involved in the optimizer and the saved master weight, momentum, and other intermediate variables all use the FP32 data type. The updated weight output by the optimizer is converted into HiF8 data for the next training iteration.

- Rounding mode selection

  As mentioned before, Float8 training is extremely sensitive to rounding. In this paper, two rounding solutions are used when data is converted to the HiF8 data type. In rounding solution 1, TA rounding is used globally. In rounding solution 2, TA rounding is used in the forward pass of training and SR is used in the backward pass of training. The training accuracy of the two rounding solutions is basically the same, except in the case of the special network YoLo_V3_Tiny and other unknown networks. Because SR has uncertainty, it is recommended that rounding solution 1 be used by default and rounding solution 2 be used only when necessary. If it is difficult to implement rounding solution 2, using SR globally can serve as an alternative. However, the training accuracy will be slightly lower than that of rounding solution 2.

HiF8 mixed precision training is highly consistent with FP16 mixed precision training in terms of the process, except that the former converts the data type of GEMM input from FP16 to HiF8 and they use different rounding solutions, as shown in Figure 3. Therefore, users can switch from the FP16 mixed precision training mode to the HiF8 mixed precision training mode, improving the training speed and energy efficiency without perceiving any difference or adding extra learning costs.

4.2 Simulation Experiment Platform for HiF8 Training and Inference

Currently, no hardware platform is available to support the HiF8 data format and complete the computation process. Therefore, the accuracy of HiF8 training on AI networks cannot be directly verified. The data range of HiF8 is a subset of the data range of FP16. Therefore, a conversion function can be used to convert FP16 numbers into a value range that can be expressed by the HiF8 data format. AI

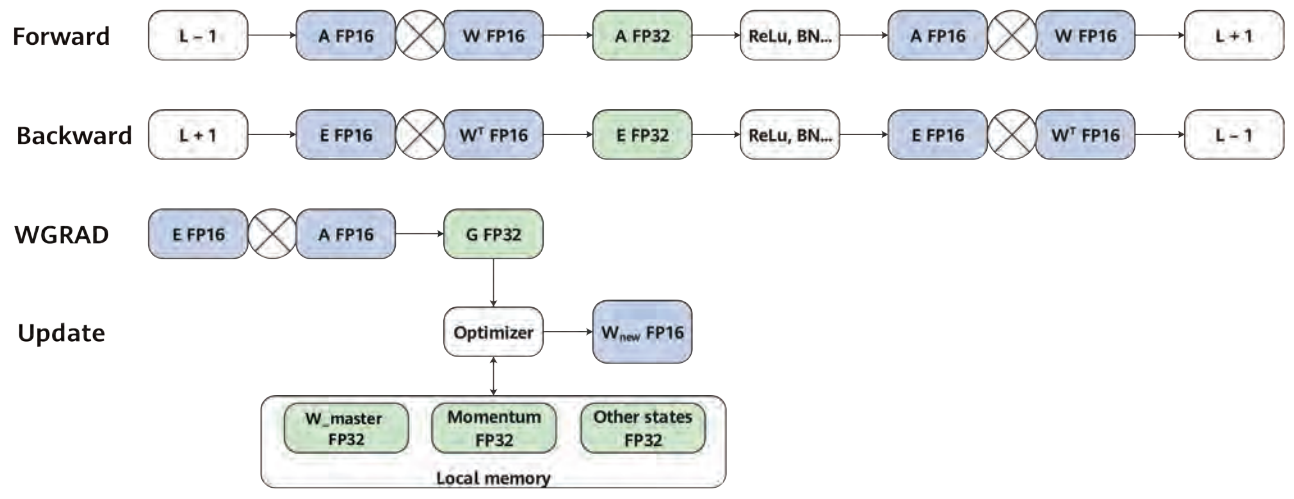


Figure 3 Process of FP16 mixed precision training

hardware platforms, such as NVIDIA GPUs and Huawei Ascend NPUs, can be utilized to conduct the simulation experiments of HiF8 training and inference on AI networks.

A data type conversion operation needs to be inserted in the original computation process of the GEMM operation in the forward pass of training, as shown in Figure 4. The HiF8 converter is inserted before the matrix multiplication operator to convert the data types of the feature map and weight from FP16 or FP32, to HiF8. The FP32 data type is used for accumulation involved in matrix multiplication. The backward pass of training involves the derivation of the feature map (*dx*) and derivation of the weight (*dw*). The data type conversion in the *dx* process is similar to that in the forward pass of training, that is, the types of gradient data and weight data are converted to HiF8 before derivation. The input data of the *dw* process is the gradient and feature map, which need to be converted into the HiF8 data type before derivation.

4.3 Scenario 1: HiF8 Training

To verify the training performance of HiF8 in the AI field, we choose two main application directions for simulation experiments: computer vision and natural language processing (NLP). The subdivided application scenarios of computer vision include classification, detection, and segmentation. In this paper, the most widely used typical networks based on the CNN and Transformer structures are selected for the experiments, including ResNet series [41], ResNext [42], Vgg [43], MobileNet [44], Inception [45], EfficientNet [46], DenseNet [47], ViT series [48], YoLo series [49], DeepLab [50], and 3D U-Net [51]. For NLP, the mainstream Transformer model [2] is used.

Figure 5 compares the loss convergence of HiF8 mixed precision training and FP16 mixed precision training (base)

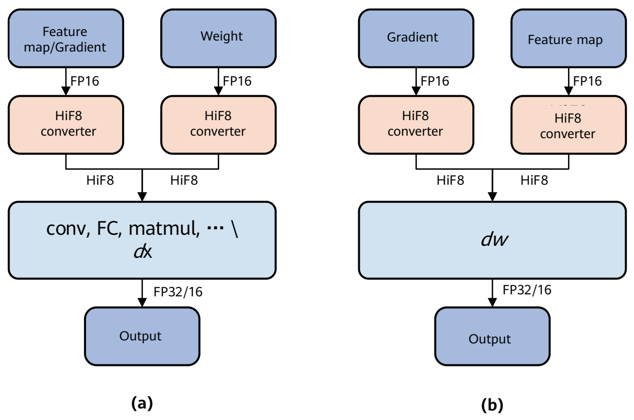


Figure 4 Simulation experiment methods for HiF8 AI training and inference

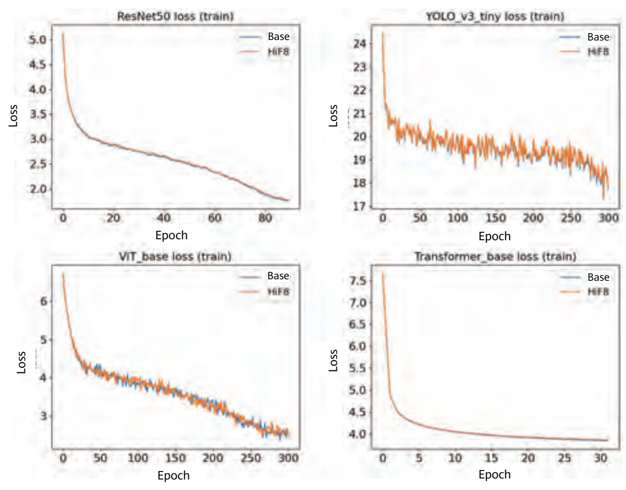


Figure 5 Convergence curves of HiF8 AI training

on the ResNet50, YoLo_V3_Tiny, ViT_base, and Transformer_base network models. The convergence curve of HiF8 is highly overlapped with that of FP16, the convergence speed of HiF8 is the same as that of FP16, and both HiF8 network training and FP16 network training can be completed within the same number of epochs.

Table 5 lists the experiment results of the training accuracy of HiF8 mixed precision, half-precision HP/FP16 mixed precision, and single-precision SP/FP32. For a fair comparison, the number of epochs of each network in different training solutions is the same. To avoid the jitter and deviation caused by the initial training value, the average value of four to six experiments is used as the training accuracy of HP and HiF8. In the CNN-based classification scenario of computer vision, the average accuracy of HiF8 mixed precision training on 12 typical networks is 0.16% lower than that of HP training and 0.15% lower than that of SP training. In Transformer-based classification and machine translation scenarios, the average accuracy of HiF8 mixed precision training on five typical networks is 0.12% higher than that of HP training

Table 5 Accuracy of HiF8 in AI training experiments

Category	Model	HiF8 Backward	SP	Mean (HP)	Mean (HiF8)	HiF8 – HP	HiF8 – SP
Classification: CNN	DenseNet121	TA	76.03%	76.04%	75.84%	–0.20%	–0.19%
	EfficientNet-b0	TA	77.02%	77.33%	77.08%	–0.25%	0.06%
	Inception-v3	SR	77.94%	77.85%	77.79%	–0.06%	–0.15%
	MobileNet-v2	TA	72.67%	72.41%	72.10%	–0.31%	–0.57%
	ResNet18	TA	70.59%	70.78%	70.64%	–0.14%	0.05%
	ResNet34	SR	74.50%	74.34%	74.24%	–0.10%	–0.26%
	ResNet50	TA	77.35%	77.36%	77.31%	–0.05%	–0.04%
	ResNet101	TA	78.92%	78.84%	78.72%	–0.12%	–0.20%
	ResNet152	SR	79.28%	79.42%	79.26%	–0.16%	–0.02%
	ResNext50_32x4d	TA	78.25%	78.14%	78.01%	–0.13%	–0.24%
	Vgg16	SR	74.10%	74.19%	73.90%	–0.29%	–0.20%
	Vgg16-BN	TA	74.59%	74.59%	74.54%	–0.05%	–0.05%
Mean_CNN						**–0.16%**	**–0.15%**
Classification: Transformer	vit_base_patch16	SR	/	79.19%	79.11%	–0.06%	/
	vit_base_patch32	SR	74.13%	74.10%	74.22%	0.12%	0.09%
	vit_large_patch16	SR	/	76.08%	76.45%	0.37%	/
	vit_large_patch_32	SR	/	71.77%	71.82%	0.05%	/
Machine Translation	Transformer-base	TA	25.93%	25.92%	26.04%	0.12%	0.11%
Mean_Transformer						**0.12%**	**0.10%**
Detection	YoLo-V3	TA	/	43.70%	43.60%	–0.10%	/
	YoLo-V3-Tiny	SR		16.63%	16.43%	–0.20%	/
Segmentation	DeepLab-V3	SR	/	78.65%	78.51%	–0.14%	/
	3D U-Net	Stops when val_acc ≥ 90.70%. Epoch number is almost the same.					
Mean_ALL						**–0.09%**	**–0.12%**

and 0.10% higher than that of SP training. For all the 20 neural networks (excluding 3D U-Net) in detection and segmentation scenarios, the average accuracy of HiF8 mixed precision training is 0.09% and 0.12% lower than that of HP and SP training, respectively. The preceding experiment results are obtained on the prerequisite that other data types are converted to HiF8 at the input layer of each network. It is a common practice in academia to use high-precision data formats for training at the input layer. If the input layer is allowed to use the HP data format, the average accuracy of HiF8 mixed precision training is 0.12% lower than that of HP training on 12 experiment networks in the CNN-based classification scenario.

AI models trained from scratch using HiF8 mixed precision can be directly used for inference.

4.4 Scenario 2: Model Retraining After Conversion from SP/HP to HiF8

Currently, most open-source AI pre-training models are obtained through HP mixed precision training or SP training. After the parameters of all layers of an existing pre-trained model are converted to the HiF8 format, the training accuracy of the original model can be reached only after a small number of epochs. In this way, a high-precision model can be quickly converted into a HiF8 mixed precision model. Table 6 describes the retraining experiment results of HiF8 mixed precision. The retraining accuracy of the HiF8 model is 0.20% higher on average than that of the original HP/SP baseline model in classification, detection, segmentation, and NLP tasks. In the generative adversarial network (GAN) scenario, the output layer has high requirements for data computing precision. After all the parameters are converted into the HiF8 format, the peak signal-to-noise ratio (PSNR) decreases during retraining. The subsequent sections about HiF8 inference and quantization will provide the model calibration accuracy achieved when the input and output layers use the HP format.

AI models retrained using HiF8 mixed precision can be directly used for inference.

5 HiF8 AI Inference

This section describes the HiF8 mixed precision inference solution. As mentioned above, AI models based on HiF8 mixed precision training and retraining can be directly used for inference. Therefore, the inference solution described in this section is mainly used to implement HiF8 mixed precision inference on networks trained through non-HiF8 (e.g., HP mixed precision and SP) mixed precision training in cases where retraining is not allowed. The following describes the HiF8 mixed precision calibration process, the Ascend hardware implementation solution for HiF8 inference calibration, and simulation experiment results in HiF8 inference scenarios where retraining is not allowed.

Table 6 Accuracy of HiF8 in AI retraining experiments

Category	Pre-trained Model	Epochs	Metric	Baseline	HiF8_Retrain	HiF8 – Baseline
Classification	ResNet50	3	Top 1 Acc	76.130%	75.816%	–0.314%
	Vgg16	3	Top 1 Acc	71.592%	71.510%	–0.082%
	Inception_v3	3	Top 1 Acc	77.212%	77.900%	0.688%
	MobileNet_v2	10	Top 1 Acc	71.878%	71.310%	–0.568%
Detection	SSD300_Vgg16 [52]	10	bbox mAP	25.070%	24.750%	–0.320%
NLP	BERT-Large-MRPC [53]	5	F1	87.437%	90.018%	2.581%
	BERT-Large-SQuADv1.1	2	F1	91.388%	90.96%	–0.425%
Segmentation	3D U-Net	5	mean_dice	90.983%	91.015%	0.032%
Mean						0.20%
GAN	ESRGAN [54]	2	PSNR	26.627	26.0256	–0.6014
Speech	Tacotron [55]	61	Loss	0.41657	0.46274	0.046171

5.1 HiF8 Calibration Process

In traditional inference calibration solutions targeted to Int8, some Int8 data with large values but small proportions needs to be saturated. The saturated Int8 data is quantized using multiplication before the GEMM operation and dequantized using multiplication after the GEMM operation. In this way, the narrow dynamic range of Int8 can be utilized to the maximum extent [56]. As a floating-point format, HiF8 has a large dynamic range of [–32768, +32768]. Therefore, data saturation processing and the fine-grained matching between scale data and a narrow dynamic range are not needed. As shown in Figure 1, HiF8 with tapered precision has relatively high precision in the exponent range [–3, +3] and sub-high precision in the exponent range [–7, +7]. Therefore, the core of the HiF8 calibration solution lies in moving the exponent of data until the exponent falls into the high-precision exponent range of the HiF8 format. To be more specific, assume that the overall exponent center of the weight data or feature map data of a specific network layer is near integer Ec. High-precision floating-point data is multiplied by 2^{-Ec} (equivalent to subtracting Ec from the exponent of the high-precision floating-point data in hardware), then converted into HiF8 format, and finally input into the GEMM for computing. The exponent moving operation needs to be restored by multiplying the FP32 or FP16 output data of GEMM by 2^{Ec} (equivalent to adding Ec to the exponent of the high-precision floating-point data in hardware). In conclusion, the core of HiF8 mixed precision calibration is to find a proper Ec value for each network layer in order to minimize the metric loss of HiF8 mixed precision inference.

This paper proposes a hierarchical calibration solution that is target metric loss–oriented based on the characteristics of the HiF8 data type and its inference performance on different types of networks. First, a metric decrease threshold needs to be specified. For example, the threshold of the accuracy decreases relative to the original networks can be set to 1% or 0.5% for evaluating whether the inference accuracy reaches the standard after calibration. Then, the hierarchical calibration process starts. The accuracy is tested once when each level of calibration is complete. If the accuracy reaches the standard, the calibration stops. Otherwise, the next level of calibration starts. Figure 6 shows the overall inference calibration process of HiF8 mixed precision. The dark purple, light blue, and light purple block diagrams indicate the level-1, level-2, and level-3 calibration processes, respectively. The red text and arrows indicate the default recommended settings for HiF8 inference calibration.

The input of the calibration process includes the following: neural network model that requires HiF8 quantization, metric type, metric loss threshold, calibration modes selected in the second and third phases, Ec search space involved in the calibration, and sensitivity statistics type used in the third phase to determine the subsequent layer skipping. The following describes the HiF8 hierarchical calibration by phase based on Figure 6.

- Phase 1: direct conversion to HiF8

  The level-1 calibration process directly converts the feature map and weight of the neural network trained through HP mixed precision or SP into the HiF8 data type, in order to form the HiF8 mixed precision model and test its accuracy. If the metric loss is less than or equal to the input threshold, the calibration process ends in advance. If the metric loss is greater than the

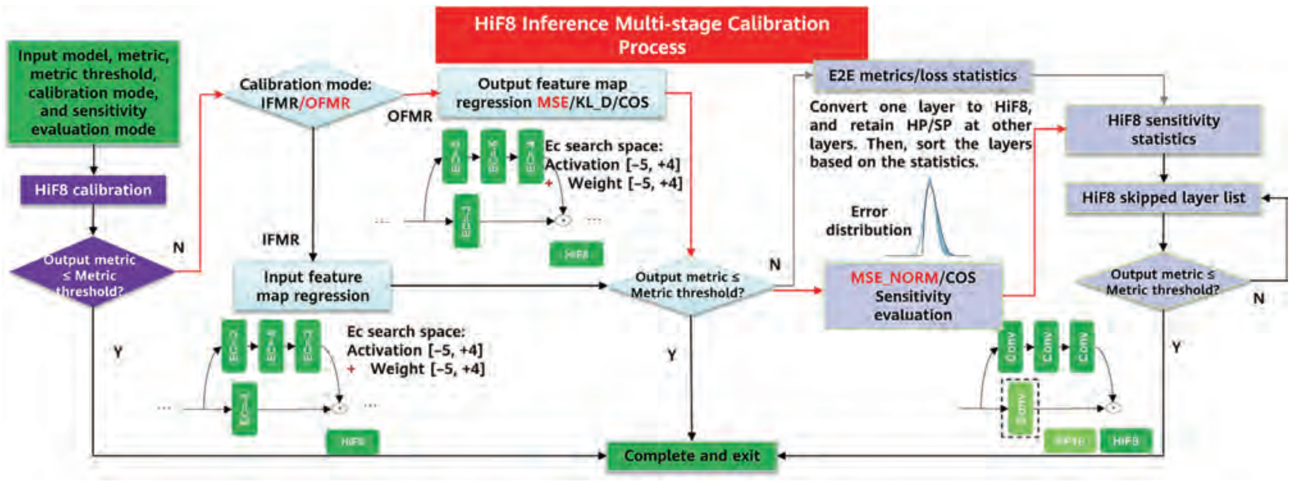

HiF8 Inference Multi-stage Calibration Process
Input model, metric, metric threshold, calibration mode, and sensitivity evaluation mode
HiF8 calibration
Output metric ≤ Metric threshold?
N
Y
Calibration mode: IFMR/OFMR
OFMR
IFMR
Output feature map regression MSE/KL_D/COS
Ec search space: Activation [–5, +4] + Weight [–5, +4]
HiF8
Input feature map regression
Ec search space: Activation [–5, +4] + Weight [–5, +4]
HiF8
Output metric ≤ Metric threshold?
N
Y
E2E metrics/loss statistics
Convert one layer to HiF8, and retain HP/SP at other layers. Then, sort the layers based on the statistics.
Error distribution
MSE_NORM/COS Sensitivity evaluation
HiF8 sensitivity statistics
HiF8 skipped layer list
Output metric ≤ Metric threshold?
N
Y
Conv
Conv
Conv
Conv
FP16
HiF8
Complete and exit

Figure 6 HiF8 hierarchical calibration process that is target metric loss–oriented

input threshold, the calibration process of the next level is triggered.

- Phase 2: direct conversion to HiF8 + calibration

HiF8 quantization is performed before data is input to the GEMM. In the quantization process, the high-precision FP32 or FP16 data is multiplied by 2^{-Ec} and then converted to the HiF8 data type. Dequantization is performed after data is output from the GEMM, that is, the output data is restored to the high-precision FP32 or FP16 data type. Therefore, the core objective of the level-2 calibration process is to find two proper Ec values for HiF8 quantization of the feature map and weight of the neural network, layer by layer, in serial mode. The lower-layer HiF8 quantization needs to inherit the result of the upper-layer HiF8 quantization. Experiments show that the optimal Ec value does not vary greatly. We recommend three input search ranges: [–5, +4], [–4, +3], and [–3, +2]. The Ec degrees of freedom (DoF_{Ec}) for the three ranges are 10, 8, and 6, respectively. You are advised not to select an excessively large Ec search space because it slows down the level-2 calibration of HiF8 inference but scarcely improves the inference accuracy. A quantization error characterization method needs to be determined to select proper Ec values. Through experiments and analysis, we conclude that the mean squared error (MSE), Kullback–Leibler divergence (KL_D), and cosine similarity (COS) methods are suitable for the HiF8 inference calibration process. MSE is recommended by default.

The level-2 calibration can be performed in two modes, depending on the types of data to which the error characterization method is applied: input feature map regression (IFMR) and output feature map regression (OFMR). As shown on the left of Figure 7, the IFMR calibration mode uses GEMM input data for regression judgment. In this case, HiF8 regression adjustment needs to be performed on the input (i.e., feature map data and weight data), to minimize the error of the two types of data after HiF8 quantization. The joint search space of IFMR is 2 × DoF_{Ec}. As shown on the right of Figure 7, the OFMR calibration mode uses GEMM output data for regression judgment. In this case, all the Ec values of both the feature map and weight need to be jointly traversed to minimize the error of the GEMM output. The joint search space of OFMR is DoF_{Ec} × DoF_{Ec}. Although the speed of OFMR calibration is lower than that of IFMR calibration, the duration of OFMR calibration is within the acceptable range. Given that the inference accuracy of OFMR is significantly higher than that of IFMR, the OFMR calibration mode is recommended by default.

After the most appropriate Ec value is selected for each layer, the inference accuracy of HiF8 level-2 calibration is tested. If the metric loss is less than or equal to the input threshold, the calibration process ends in advance. If the metric loss is greater than the input threshold, the calibration process of the next level is triggered.

- Phase 3: direct conversion to HiF8 + calibration + layer skipping

In the first two levels of calibration processes, the feature map and weight of each network layer are directly converted to HiF8, or HiF8 quantization is

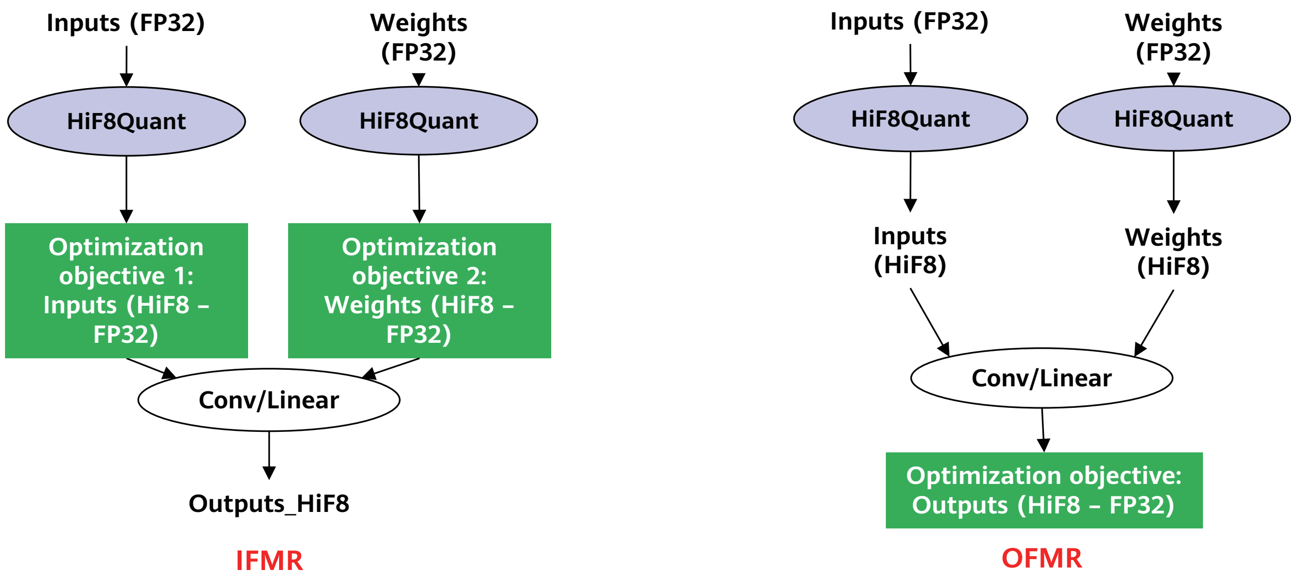


Figure 7 IFMR and OFMR calibration modes

performed on such data radically. However, the inference accuracy is not required to reach the standard. The level-3 calibration process is the last line of defense in the HiF8 hierarchical calibration solution. The inference accuracy must reach the preset target level in this process. To achieve this goal, the sensitivity of each feature map and weight that require HiF8 quantization to the overall error or network-wide inference accuracy needs to be evaluated, and a sensitivity ranking list needs to be generated. Then, the feature maps and weights with high sensitivity (i.e., which severely affect the network-wide inference accuracy) are skipped one by one and HiF8 quantization is not performed on them until the network inference accuracy reaches the target level.

- Solution 1: Evaluate the impact of the feature map and weight of each layer on the inference metrics of the entire network after HiF8 quantization in an E2E manner. As shown on the left of Figure 8, the convolution operation in the dashed-line box has a greater impact on the target accuracy than the other three convolution operations, and therefore has a higher ranking in the sensitivity ranking list. As shown on the right of Figure 8, the convolution block in the dashed-line box is skipped after calibration, and HiF8 quantization is not performed on this block, ensuring that the model inference accuracy is improved and close to the target accuracy level.

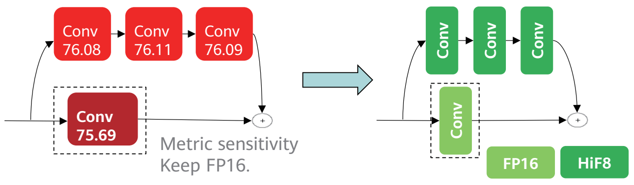


Figure 8 E2E HiF8 quantization sensitivity evaluation

- Solution 2: Reuse the error characterization information of each network layer generated in the level-2 calibration process. This solution can quickly implement sensitivity evaluation and sorting after HiF8 quantization is performed on the feature map and weight of each network layer without re-computing errors. Then, the HiF8 quantization process is skipped layer by layer in descending order of the sensitivity based on the ranking list to improve the model inference accuracy until the accuracy reaches the preset target.

Experiments show that on most networks, the number of network layers for which HiF8 quantization is skipped is the same in the two solutions. For a few networks that are difficult to quantify, only one more layer is skipped in solution 2 than in solution 1. As the calibration based on solution 1 takes a long time, solution 2 is recommended for sensitivity statistics collection by default.

5.2 Ascend Hardware Solution for HiF8 Inference Calibration

To meet the hardware requirements of HiF8 inference calibration, the next-generation Ascend chip is expected to support the solution shown in Figure 9. The HiF8 quantization and dequantization operations reuse the existing computing logic and are deployed in different processing units based on whether batch normalization (BN) is available or whether BN is fused. As shown on the top of Figure 9, in the non-fused BN scenario, the AI Vector processing unit performs HiF8 quantization on feature maps, and then FixPipe performs HiF8 dequantization on feature maps and weights. As a new processing unit of the next-generation Ascend chip, FixPipe immediately follows the Cube output and performs simple associated

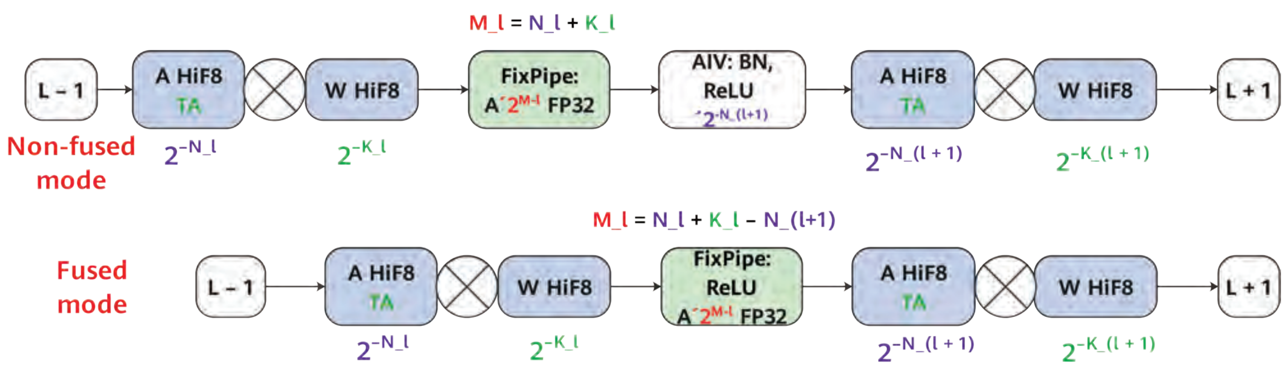


Figure 9 Ascend HiF8 inference hardware support solution

computation. As shown on the bottom of Figure 9, in the fused BN scenario, the FixPipe processing unit can simultaneously perform HiF8 dequantization on the feature map and weight of the previous network layer and perform HiF8 quantization on the feature map of the next network layer in fused mode at a time.

5.3 Scenario 3: Model Retraining Not Allowed After Conversion from SP/HP to HiF8

To verify the inference performance of the HiF8 format in the AI field, we also chose numerous neural networks in two directions: computer vision and NLP. The experiment platform has been introduced in section 4 "HiF8 AI Training." In this paper, the metric loss threshold in the HiF8 hierarchical calibration process is set to 0.5%. Table 7 lists the experiment results.

- The purple text indicates that the accuracy of the corresponding neural network reaches the standard after level-1 calibration. All Transformer networks, including the ViT series [48] and BERT_Large series [53] networks, can be used for inference immediately after direct conversion to HiF8. Therefore, the HiF8 data type can implement online conversion from training models to inference models on Transformer networks. For a few CNNs, including MaskRCNN [57] and 3D U-Net [51], the accuracy can also reach the standard after direct conversion to HiF8.
- The blue text indicates that the accuracy of the corresponding neural network can reach the standard after level-2 calibration. Most CNNs, including ResNet series [41], DenseNet [47], ResNext [42], Vgg [43], SSD [52], and YoLo-V3 [49] networks, can be used for inference after direct conversion to HiF8 followed by calibration. For Transformer networks, the inference accuracy can be further improved or even surpass the accuracy of the original model after level-2 calibration.
- The red text indicates that the accuracy of the corresponding neural network can reach the standard after level-3 calibration. A small number of CNNs on which it is difficult to perform quantization and inference, including EfficientNet_B0 [46], Inception_V3 [45], and ESRGAN [54], can be used for inference after direct conversion to HiF8 + calibration + layer skipping. The Tacotron2 [55] network based on a hybrid structure of LSTM + CNN is competent for inference after the HP data format is retained at the input and output layers. Experiments show that the identified layers to be skipped in the level-3 HiF8 inference calibration process are generally located at the front end and tail end of DNNs. These layers have a small number of weight parameters and have a limited impact on HiF8 inference acceleration.

6 Ascend HiF8 Deployment Evaluation

As described above, the current HiF8 mixed precision AI training and inference solutions mainly change the input data type of the GEMM, enabling convenient hardware implementation. The Ascend DaVinci AI Core implements GEMM computing in inner product mode [19]. Assuming that the size of matrix multiplication is (M, K) × (K, N), a single computing slice of the Cube processing unit can complete the MAC operation on 16 pairs of FP16 input data at a time, that is, K equals 16. Compared with FP16, HiF8 requires lower computing resource overhead. The next-generation Ascend chip is expected to configure the Cube computing power of HiF8 to twice that of FP16, that is, K equals 32. The MAC operation is performed on 16 pairs of HiF8 input data at a low cost by multiplexing the existing FP16 data path, as shown in Figure 10. The MAC operation on the other 16 pairs of HiF8 input data can be implemented using independent hardware. The following roughly evaluates the overhead and benefits of HiF8 hardware implementation. Accurate data will be provided after subsequent tests.

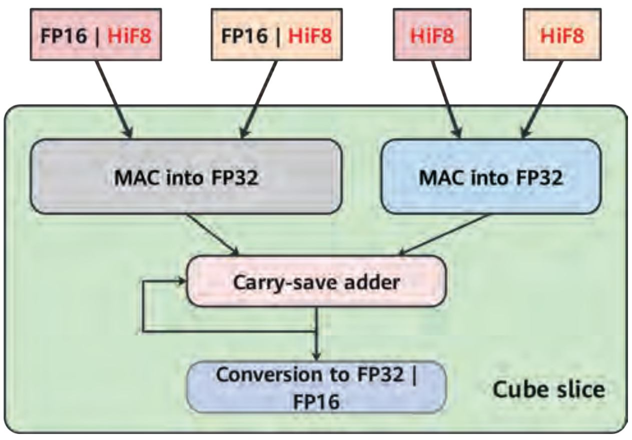


Figure 10 HiF8 Cube implementation solution

Table 7 Experimental accuracy of HiF8 AI inference calibration

Category	Model	Metric	SP	HP	HiF8	HiF8 – SP/HP	HiF8CaliNS	HiF8CaliNS – SP/HP	HiF8CaliS	HiF8CaliS – SP/HP	SkipLayer (HP)	Int8	Int8 – SP/HP
Classification	ResNet18	Top 1	69.76	/	68.98	–0.78	69.48	–0.28	/	/	/	69.61	–0.15
	ResNet34	Top 1	73.31	/	72.56	–0.76	72.96	–0.35	/	/	/		
	ResNet50	Top 1	76.13	/	74.85	–1.28	75.44	–0.69	75.93	–0.20	1/54	76.18	0.05
	ResNet101	Top 1	77.37	/	76.48	–0.89	76.89	–0.48	/	/	/	77.36	–0.01
	ResNet152	Top 1	78.31	/	77.76	–0.55	77.96	–0.35	/	/	/	78.28	–0.03
	DenseNet121	Top 1	74.43	/	73.58	–0.86	74.14	–0.29	/	/	/	74.38	–0.06
	ResNext50_32x4d	Top 1	77.62	/	76.74	–0.88	77.17	–0.44	/	/	/	77.57	–0.04
	Vgg16	Top 1	71.59	/	70.98	–0.61	71.28	–0.31	/	/	/	71.62	0.03
	Vit_base_patch32_224	Top 1	75.92	/	75.66	–0.26	75.85	–0.07	/	/	/		
	Vit_base_patch16_224	Top 1	81.07	/	80.86	–0.21	80.91	–0.15	/	/	/		
	Vit_large_patch32_224	Top 1	76.96	/	76.83	–0.13	76.87	–0.09	/	/	/		
	Vit_large_patch16_224	Top 1	79.68	/	79.69	0.00	79.77	0.09	/	/	/		
	EfficientNet_b0	Top 1	76.44	/	74.42	–2.02	74.49	–1.95	76.09	–0.35	4/82		
	EfficientNet_b0	Top 1	/	77.68	71.02	–6.66	75.16	–2.52	77.27	–0.42	5/82	74.33	–3.35
	Inception_V3	Top 1	77.92	/	76.67	–1.25	77.38	–0.54	77.69	–0.23	1/98		
	Inception_V3	Top 1	/	77.32	74.11	–3.21	76.07	–1.25	77.02	–0.30	1/98		
	MobileNet_v2	Top 1	72.40	/	69.63	–2.77	70.46	–1.93	71.97	–0.43	5/53		
	MobileNet_v2	Top 1	/	71.878	65.262	–6.62	66.80	–5.08	71.39	–0.49	7/53	71.27	–0.60
	MaskRCNN	bbox	37.8	/	37.1	–0.7	37.3	–0.50	/	/	/	37.80	0.00
Detection		segm	34.5	/	34	–0.5	34.2	–0.30	/	/	/	34.51	0.01
	SSD-Vgg16	bbox	25.1	/	24.2	–0.9	24.8	–0.30	/	/	/	25.02	–0.08
	YoLo-V3	bbox	/	43.3	42.2	–1.1	42.8	–0.5	/	/	/	43.20	–0.10
Segmentation	3D-Unet	mean_dice	/	90.983	91.027	0.044	/	/	/	/	/	91.01	0.03
NLP	Bert_Large_MRPC	F1	/	87.4372	87.188	–0.25	87.63	0.19	/	/	/	87.93	0.49
	Bert_Large_SQuADv1.1	F1	/	91.4045	91.393	–0.01	/	/	/	/	/	91.38	–0.02
Speech	Tacotron2	Loss	/	0.45389	0.49959	0.046	0.481962025	0.028069556	0.450678	0.00	2/16	0.45	0.00
GAN	ESRGAN	PSNR	26.627	/	21.6064	–5.0206	23.3981	–3.2289	26.1619	–0.47	2/351	26.49	–0.13

6.1 Area Evaluation

We evaluated the area overhead of HiF8 by using the register-transfer level (RTL) pilot code based on the microarchitecture of the Cube processing unit in Ascend *XXX*1 when K equals 32 and the chip manufacturing process is 12 nm. The experiment results show that the area of Cube computing slices increased by 14.69% after the slices supported HiF8. In Ascend *XXX*1, the area of Cube computing slices accounts for approximately 80% of the total area of Cube computing clusters, and 1% of the computing cluster area is used for HiF8 format decoding. Therefore, the area of Cube computing clusters increased by 12.75% (14.69% × 0.8 + 1%). The area of Cube computing clusters accounts for 32.7% of the total area of AI Core. In addition, 0.3% of the total area needs to be added on the Vector and FixPipe processing units for HiF8 format conversion. Therefore, when Ascend *XXX*1 serves as the baseline, the area overhead for supporting HiF8 AI training and inference is estimated to be approximately 4.5% (12.75% × 0.327 + 0.3%) of the total area of AI Core. Figure 11 shows the evaluation result. During the implementation of Ascend *XXX*2 in the future, we will consider making up for such area overhead from other aspects.

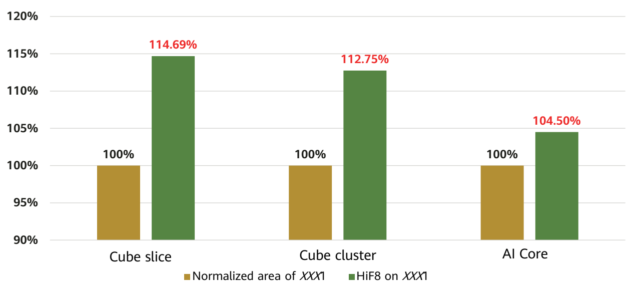


Figure 11 HiF8 normalized area overhead (12 nm manufacturing process)

6.2 Power Consumption Evaluation

We tested the Cube power consumption (K equals 16) of HiF8 on the FP16 multiplexing path based on the preceding pilot code. The mantissa bit width of HiF8 is smaller than that of FP16, and the multiplier input data on the multiplexed path has many zero bits. Therefore, power-saving gains can be achieved along the multiplexed path. Experiments using real data as test cases show that the Cube power consumption measured when the multiplexed path uses HiF8 input data was approximately 71.3% of that measured when the multiplexed path uses FP16 input data. Taking a single Cube processing unit for which the FP16 computing power is 4096 (163) as the baseline, the power consumption of FP16 mixed precision is 14.14 W. The power consumption of HiF8 on the multiplexed path is 10.08 (14.14 × 71.3%) W. The power consumption of the other HiF8 path with independent hardware overhead (K equals 16) was calculated to be approximately 3.67 W. Therefore, the total Cube power consumption of HiF8 mixed precision (K equals 32) is approximately 13.75 (10.08 + 3.67) W. Table 8 compares the power consumption overheads of a single Cube unit in various computing formats. In terms of training, the power consumption of HiF8 is equivalent to that of FP16. In terms of inference, the power consumption of HiF8 is slightly higher than that of Int8. Both training and inference will be considered in the implementation to properly balance the area, power consumption, and performance.

Table 8 Evaluated power consumption of the Cube unit (12 nm manufacturing process)

Format: Computing Powerntional Prefix Code	FP16: 16^3 = 4096	HiF8: 16 × 32 × 16 = 8192	Int8: 16 × 32 × 16 = 8192
Power Consumption	14.14 W	13.75 W	11.43 W

6.3 Training Performance Evaluation

In the paper, the ESL models of Ascend *XXX*1 and *XXX*2 are properly modified to evaluate the time benefits of HiF8 mixed precision training based on simulation experiments. Table 9 describes the configurations of the two ESL models. Note that the ESL model of Ascend *XXX*2 is a phased model rather than the final version.

Table 9 ESL model configurations of Ascend *XXX*1 and *XXX*2

ESL Model	**Freq. (GHz)**	**#Core**	**L2 Cache (MB)**	**L2 BW (B/s)**	**HBM (TB)**	**Ring Freq. (GHz)**
*XXX*1	1.5	24	192	6.4	1.6	2.8
*XXX*2	1.2	24	150	5.2	1.92	1.9

In this paper, two typical representative network models were selected as experiment samples: CNN-based ResNet50 [41] and Transformer-based BERT [53]. The forward pass and backward pass of network-wide training were conducted on ResNet50, and those of one block (24 identical blocks in total) were conducted on BERT. Table 10 lists the experiment results.

HiF8 mixed precision training (K equals 32) leads to the

Table 10 Details about HiF8 training performance evaluation

Comparison Item		***XXX*1 ESL Model**		***XXX*2 ESL Model**	
		ResNet50	**BERT**	**ResNet50**	**BERT**
Read Bytes	**FP16**	13075332912	11041507200000	13075332912	11041507200000
	HiF8	9773923104	6684673920000	9773923104	6684673920000
Read Reduction		25%	39%	25%	39%
Write Bytes	**FP16**	6189563616	2560819440000	6189563616	2560819440000
	HiF8	5330673360	2639462640000	5330673360	2639462640000
Write Reduction		14%	–3%	14%	–3%
Cube Cycles	**FP16**	5850340	5486670	7450100	6630180
	HiF8	2257750	3316480	4422080	3832810
Cube Reduction		**61%**	**40%**	**41%**	**42%**
Vector Cycles	**FP16**	5257550	2435720	6176530	2711960
	HiF8	5314640	2532780	6154260	2717400
Vector Reduction		–1.08%	–3.98%	0.36%	0.20%
Total Cycles	**FP16**	8136272	5588293	9972964	6750466
	HiF8	6481540	3467137	7587942	4053308
Total Reduction		**20%**	**38%**	**24%**	**40%**
FP16 Cube/Vector		1.11	2.25	1.21	2.44
HiF8 Cube/Vector		0.42	1.31	0.72	1.41

following changes compared with FP16 mixed precision training (K equals 16) after Cube supports HiF8 input:

- The amount of data in the interaction between AI Core and the out-of-core high bandwidth memory (HBM) decreased dramatically.
- The time for Cube to complete GEMM computing tasks (including data read and storage) was significantly reduced.
- The time for Vector to complete computing tasks (including data read and storage) basically remained unchanged.

As shown in Figure 12, the experiment results show that the training performance of HiF8 mixed precision can be improved by 26% and 31% on the ResNet50 network and by 61% and 67% on the BERT network for the ESL models

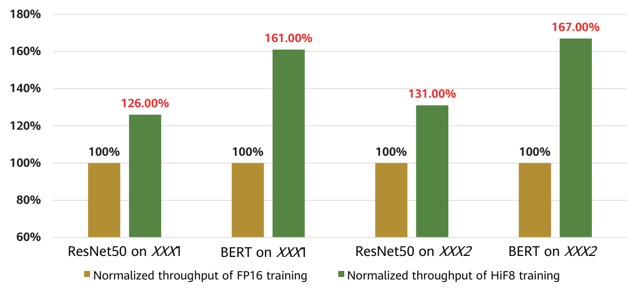


Figure 12 Performance improvement of HiF8 normalized training

of Ascend *XXX*1 and *XXX*2 respectively, when compared with FP16 mixed precision. The main reason for the training performance improvement on ResNet50 being lower than that on BERT was that the network performance bottleneck of ResNet50 changed from Cube Bound to Vector Bound, whereas the performance bottleneck of BERT remained Cube Bound after HiF8 mixed precision was used.

7 Comparison Between Ascend HiF8 and Hopper FP8

This section compares the similarities and differences between NVIDIA Hopper H100 with FP8 mixed precision [33] and Ascend HiF8 mixed precision. Hopper FP8 has two Float8 data formats with fixed field bit widths. The FP8 (1-4-3) format favors computing precision, whereas the FP8 (1-5-2) format favors dynamic range. As a single Float8 data format, Ascend HiF8 balances the computing precision and dynamic range. The essential differences in data representation capabilities determine the differences between the two formats in software and hardware deployment and user-friendliness, as described in Table 11. It can be concluded that Ascend HiF8 is simpler and easier to use than Hopper FP8.

Table 11 Comparison between Hopper FP8 and Ascend HiF8

Hopper FP8	Ascend HiF8
Two formats: FP8 (1-4-3) and FP8 (1-5-2)	Single data format
Poor generalization. The computing precision and dynamic range cannot be balanced.	Strong generalization. The computing precision and dynamic range are balanced.
Application in matrix multiplication: Tensor Core (TC) supports corresponding computations.	Application in matrix multiplication: Cube Core supports corresponding computations.
With the same computing power as FP16, the amount of GEMM input data is halved.	With the same computing power as FP16, the amount of GEMM input data is halved.
The training process is different from that of the existing FP16 + FP32 mixed precision. The data format needs to be selected, and the scaling value needs to be determined.	The training process is the same as that of the existing FP16 + FP32 mixed precision. The format is user-friendly, easy to use, and users are not required to know its details.
Transformer Engine is required for analyzing TC output statistics and determining the data format and scaling value.	No extra hardware overhead, statistics collection, data format selection, or scaling is required.
The training accuracy on Transformer networks is the same as that of FP16. Visual tasks are supported, but the neural network type is not open to public. The accuracy of the parameter-efficient model decreases by 1%.	The training accuracy on Transformer networks is slightly higher than that of FP16. The training accuracy of CNN models decreases by 0.16% on average, and by a maximum of 0.31%.
The Transformer model obtained through training can be directly used for FP8 inference without fine-tuning.	The Transformer or CNN model obtained through training can be directly used for HiF8 inference without fine-tuning. The Transformer model obtained through SP or HP training can be used for inference without calibration after direct conversion to HiF8.

8 Conclusion and Outlook

This paper proposes the innovative HiF8 data format, HiF8 AI training solution, and HiF8 AI inference solution. Experiments show that HiF8 mixed precision has high training and inference accuracy on mainstream DNNs based on CNN and Transformer structures. This paper also evaluates the overhead and benefits of deploying the HiF8 AI solution on Ascend chips. According to the evaluation based on Ascend *XXX*1, HiF8 mixed precision can improve the training performance on ResNet50 and BERT by 26% and 61%, respectively, with an AI Core area penalty of 4.5% and power consumption close to that of FP16 mixed precision training.

In the future, while maintaining the user friendliness of HiF8 mixed precision training, we will continue to explore ways to further improve the training accuracy on CNNs. Currently, some progress has been made in this direction. Experiments show that HiF8 mixed precision can achieve computing precision convergence for HPL_AI. Therefore, we will also consider exploring the possibility of applying HiF8 mixed precision to HPC scenarios in the future.

As one of the key features and core competencies of the next-generation Ascend chip, HiF8 AI mixed precision will further enable Huawei to realize the vision of "Bringing digital to every person, home and organization for a fully connected, intelligent world".

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, "ImageNet classification with deep convolutional neural networks," Advances in Neural Information Processing Systems, 25 (2012).

[2] Ashish Vaswani *et al.*, "Attention is all you need," *Advances in Neural Information Processing Systems*, 30 (2017).

[3] Chris A Mack, "Fifty years of Moore's law," *IEEE Transactions on Semiconductor Manufacturing*, 24.2 (2011): 202–207.

[4] Seppo Linnainmaa, "Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä," Diss. Master's thesis, University of Helsinki, 1970.

[5] John J Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, 79.8 (1982): 2554–2558.

[6] Institute of Electrical and Electronics Engineers, Computer Society Standards Committee, and David Stevenson, "IEEE standard for binary floating-point arithmetic," *IEEE*, 1985.

[7] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh, "A fast learning algorithm for deep belief nets," *Neural Computation*, 18.7 (2006): 1527–1554.

[8] Kumar Chellapilla, Sidd Puri, and Patrice Simard, "High performance convolutional neural networks for document processing," *Tenth International Workshop on Frontiers in Handwriting Recognition*, Suvisoft, 2006.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David, "BinaryConnect: Training deep neural networks with binary weights during propagations," *Advances in Neural Information Processing Systems*, 28 (2015).

[10] Joachim Ott *et al.*, "Recurrent neural networks with limited numerical precision," arXiv preprint arXiv: 1608.06902 (2016).

[11] Itay Hubara *et al.*, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, 18.1 (2017): 6869–6898.

[12] Shuchang Zhou *et al.*, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," arXiv preprint arXiv: 1606.06160 (2016).

[13] Suyog Gupta *et al.*, "Deep learning with limited numerical precision," *International Conference on Machine Learning*, PMLR, 2015.

[14] TensorFlow Preliminary White Paper, http://download.tensorflow.org/paper/whitepaper2015.pdf, 2015.

[15] Martín Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," arXiv preprint arXiv: 1603.04467 (2016).

[16] Dhiraj Kalamkar, *et al.*, "A study of BFLOAT16 for deep learning training," arXiv preprint arXiv: 1905.12322 (2019).

[17] Dan Zuras, *et al.*, "IEEE standard for floating-point arithmetic," IEEE Standard 754.2008 (2008): 1–70.

[18] NVIDIA Tesla V100 GPU Architecture, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[19] Heng Liao *et al.*, "DaVinci: A scalable architecture for neural network computing," *Hot Chips Symposium*, 2019.

[20] Paulius Micikevicius *et al.*, "Mixed precision training," arXiv preprint arXiv: 1710.03740 (2017).

[21] Ankur Agrawal *et al.*, "DLFloat: A 16-b floating point format designed for deep learning training and inference," *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, IEEE, 2019.

[22] NVIDIA A100 Tensor Core GPU Architecture, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[23] John L. Gustafson, "A radical approach to computation with real numbers," *Supercomputing Frontiers and Innovations*, 3.2 (2016): 38–53.

[24] John L. Gustafson and Isaac T. Yonemoto, "Beating

floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, 4.2 (2017): 71–86.

[25] Jinming Lu *et al.*, "Training deep neural networks using posit number system," *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, IEEE, 2019.

[26] Naigang Wang, *et al.*, "Training deep neural networks with 8-bit floating point numbers," *Advances in Neural Information Processing Systems*, 31 (2018).

[27] Xiao Sun *et al.*, "Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks," *Advances in Neural Information Processing Systems*, 32 (2019).

[28] Ankur Agrawal *et al.*, "A 7 nm 4-core AI chip with 25.6 TFLOPS hybrid FP8 training, 102.4 TOPS INT4 inference and workload-aware throttling," *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 2021.

[29] Sean Fox *et al.*, "A block minifloat representation for training deep neural networks," *International Conference on Learning Representations*, 2020.

[30] Jeongwoo Park, Sunwoo Lee, and Dongsuk Jeon, "A neural network training processor with 8-bit shared exponent bias floating point and multiple-way fused multiply-add trees," *IEEE Journal of Solid-State Circuits* (2021).

[31] Tesla Dojo Technology, https://tesla-cdn.thron.com/static/SBY4B9_tesla-dojo-technology_OPNZ0M.pdf, 2021.

[32] Introducing AWS Trainium-based Amazon EC2 Trn1 instances, https://www.youtube.com/watch?v=_6y-AWmTako, 2021.

[33] NVIDIA H100 Tensor Core GPU Architecture, https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf, 2022.

[34] Sophia Susan Raju *et al.*, "Application of noise to avoid overfitting in TCAD augmented machine learning," *2020 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, IEEE, 2020.

[35] Chuan-Jia Jhang *et al.*, "Challenges and trends of SRAM-based computing-in-memory for AI edge devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68.5 (2021): 1773–1786.

[36] Gonçalo Raposo, Pedro Tomás, and Nuno Roma, "PositNN: Training deep neural networks with mixed low-precision posit," *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE, 2021.

[37] Jinming Lu *et al.*, "Evaluations on deep neural networks training using posit number system," *IEEE Transactions on Computers*, 70.2 (2020): 174–187.

[38] Rajesh Bordawekar, Bülent Abali, and Ming-Hung Chen, "EFloat: Entropy-coded floating point format for deep learning," arXiv preprint arXiv: 2102.02705 (2021).

[39] Michael P. Connolly, Nicholas J. Higham, and Theo Mary, "Stochastic rounding and its probabilistic backward error analysis," *SIAM Journal on Scientific Computing*, 43.1 (2021): A566–A585.

[40] Pierre L'Ecuyer, "History of uniform random number generation," *2017 Winter Simulation Conference (WSC)*, IEEE, 2017.

[41] Kaiming He *et al.*, "Deep residual learning for image recognition," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[42] Saining Xie *et al.,* "Aggregated residual transformations for deep neural networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[43] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv: 1409.1556 (2014).

[44] Mark Sandler *et al.*, "MobileNetV2: Inverted residuals and linear bottlenecks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[45] Christian Szegedy *et al.*, "Rethinking the inception architecture for computer vision," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[46] Mingxing Tan and Quoc V. Le, "EfficientNet: Rethinking model scaling for convolutional neural

networks," *International Conference on Machine Learning, PMLR*, 2019.

[47] Gao Huang *et al.*, "Densely connected convolutional networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[48] Alexey Dosovitskiy *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," arXiv preprint arXiv: 2010.11929 (2020).

[49] Joseph Redmon and Ali Farhadi, "YOLOv3: An incremental improvement," arXiv preprint arXiv: 1804.02767 (2018).

[50] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam, "Rethinking atrous convolution for semantic image segmentation," *Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE/CVF, Vol. 6. 2017.

[51] Özgün Çiçek *et al.*, "3D U-Net: Learning dense volumetric segmentation from sparse annotation," *International Conference on Medical Image Computing and Computer-assisted Intervention*, Springer, Cham, 2016.

[52] Wei Liu *et al.*, "SSD: Single shot multibox detector," *European Conference on Computer Vision*, Springer, Cham, 2016.

[53] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "BERT: Pre-training of deep bidirectional Transformers for language understanding," *Proceedings of NAACL-HLT*, 2019.

[54] Xintao Wang *et al.*, "ESRGAN: Enhanced super-resolution generative adversarial networks," *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*. 2018.

[55] Yuxuan Wang *et al.*, "Tacotron: Towards end-to-end speech synthesis," arXiv preprint arXiv: 1703.10135 (2017).

[56] 8-bit Inference with TensorRT, https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf, 2017.

[57] Kaiming He *et al.*, "Mask R-CNN," *Proceedings of the IEEE International Conference on Computer Vision*, 2017.

DP SQRT Computation Principle and Ultra-Low Latency Microarchitecture Design

Yuanyong Luo [*], Zichao Long, Zhiyan Gu, Jianfeng Wang
HiSilicon Semiconductor and Component Business Dept

Abstract

In computationally intensive tasks, the complex CPU instruction for floating-point square root (FP SQRT) computation often becomes a bottleneck that limits the performance. Based on polynomial fast approximation and indirect error accurate rounding, this paper innovatively constructs an FP SQRT computation precision doubling method with high and low bits separated, and proposes a new double precision floating-point square root (DP SQRT) computation principle and its corresponding microarchitecture design. Compared with the existing Sweeney-Robertson-Tocher (SRT) algorithm, the proposed design can reduce the latency by 47% and increase the throughput by 9 times, significantly improving performance.

* Corresponding author

1 Overview

FP SQRT computation, as one of the basic operations, has developed into a complex arithmetic logic unit (ALU) that must be supported by advanced CPUs. In particular, DP SQRT has the highest requirements for computation precision and is the most difficult to implement on hardware with high performance. The earliest FP SQRT implementation on hardware is bitwise iteration [1, 2] based on the manual computation process. This method accurately computes one bit in each iteration, leading to high latency. Therefore, this method has been phased out from the industry. Researchers are seeking various low-latency strategies [3–8] based on bitwise iteration. Among them, the SRT iteration method has the most far-reaching influence [9, 10]. With this method, multiple bits are accurately computed in each iteration, and the final significant bit width is accumulated through multiple iterations. The more bits generated by each SRT iteration, the fewer the number of iterations. However, the complexity of selecting partial results during iteration limits the size of the radix. Currently, only SRT is used, with the radix up to 2^8. That is, a maximum of 8 bits are computed in each iteration. In this case, the latency is still high for DP SQRT. To address this problem, researchers propose an SRT-based very-high-radix design method [6]. Its principle is to pre-scale the radicand first so that the radical result is closer to the final SQRT computation result. Then the logic of selecting partial results during SRT iteration becomes simple. According to research, a polynomial approximation pre-scaling factor can be used to increase the number of bits generated by each SRT iteration to 12.

The computation precision of SRT iteration is linearly converged. There are also some iteration methods that adopt quadratic convergence, such as Newton's method [11, 12], Goldschmidt’s algorithm [13, 14], and Babylonian algorithm [15, 16]. Although these methods have a faster convergence speed, the final latency depends on the precision of the initial estimate. If the precision of the initial estimate is not high, the latency is high. The biggest problem with these methods is that the error is difficult to control after multiple iterations. Currently, no accurate rounding can be implemented based on these methods, and hence these methods cannot meet the rounding requirements of the IEEE 754 [17] international standard.

This paper proposes a variant of the Babylonian algorithm and innovatively constructs an SQRT computation method with high and low bits separated, to greatly reduce the area and latency in iterative computation. For DP SQRT computation, we first use piecewise second-order polynomial fitting to achieve half of the required precision of DP SQRT at a time, that is, the high bits of DP SQRT are computed. Then, an iteration is performed using the variant of the Babylonian algorithm, which directly computes the low bits of DP SQRT. The high and low bits are added to obtain the full-precision result of DP SQRT. Finally, after strict error analysis and control, we propose the corresponding accurate rounding method, which meets the IEEE 754 standard.

The DP SQRT computation method proposed in this paper has a throughput of 1 sample per cycle, a latency of only 7 to 8 cycles, and a performance far beyond the most advanced very-high-radix SRT iteration method.

2 Piecewise Polynomial Approximation

For complex unary functions, such as transcendental functions ln, exp, sin, and cos, and non-transcendental functions, such as div and sqrt, piecewise polynomial approximation [18] is a common LUT-based low-latency hardware implementation method. It can be classified into uniform piecewise polynomial approximation [19, 20], non-uniform piecewise polynomial approximation [21, 22], and error-flattened piecewise polynomial approximation [23, 24]. Among them, uniform piecewise polynomial approximation has the simplest method for indexing polynomial coefficients and is introduced as the basic design component in this paper.

Assume that the target computation function is $f(X)$, where X is a T-bit fixed-point number. Uniform piecewise polynomial approximation divides T-bit X into address a with higher U bits and x with lower L (T minus U) bits, where $x = [0, 1)$. Address a partitions the target function $f(X)$ into 2^U segments $f_i(X)$ at equal intervals. Then, in each segment, $f_i(X)$ is uniformly mapped to the low-bit x, and $p_i(x)$ is used for polynomial fitting, with the aim to solve the coefficient of each piecewise polynomial $p_i(x)$ and minimize the maximum absolute error (MAE).

$$min\left(\max|f_i(X) - p_i(x)|\right)$$

Letting $p_i(x)$ be an n-order polynomial, Chebyshev's theorem [25] proves that when and only when $n + 2$ values exist:

$$0 \le \mathrm{x}_0 < x_1 < ... < x_{n+1} < 1$$

The following formula is true:

$$|p_i(x_i) - f_i(X_i)| = max\,|p_i(x_i) - f_i(X_i)|$$

Fitting $f_i(X)$ by polynomial $p_i(x)$ achieves the best performance. That is, for n-order polynomial fitting, the fitting curve and target curve can generate a maximum of $n+1$ intersection points, and further result in $n+2$ extreme points. The MAE of curve fitting is the smallest when and only when the absolute errors of the $n+2$ extreme points are equal.

Based on Chebyshev nodes [26], a unary function fitting tool Poly-K suitable for any K-order piecewise polynomial is developed to achieve the best fitting performance. The Poly-K tool consists of a segmenter and a quantizer. The segmenter ensures that the MAE_{sw} of the polynomial fitting the target function is less than $0.5\ ulp$ in the case of the minimum number of segments (that is, the minimum U value) at high precision FP64. The quantizer quantizes the polynomial coefficient of each segment expressed by FP64 and the intermediate results of polynomial computation into fixed-point numbers, and ensures that the MAE_{hw} of target function fitting computed by hardware is less than $1\ ulp$. The left part of Figure 1 shows an error curve of fitting the target function by using a 2-order polynomial by the segmenter. Each segment fitting curve and the target function have three intersection points and four extreme points, and the absolute errors of the four extreme points are equal. In this case, MAE_{sw} is less than $0.5\ ulp$. The right part of the figure shows an error curve obtained after the quantizer performs fixed-point quantization on the floating-point computation results, where MAE_{hw} is less than $1\ ulp$. The Poly-K tool and Remez algorithm [27, 28] can achieve basically the same error performance in piecewise polynomial fitting of complex functions.

In the following sections, Poly-K is used as a basic tool to achieve 1-ulp error computation for complex functions.

3 DP SQRT Algorithm

For conventional iteration precision doubling algorithms, such as Newton's method, Goldschmidt's algorithm, and Babylonian algorithm, the precision bit width cost that was computed last time needs to be included in the next iterative computation. For example, if k-bit precision is output in the first iteration, 2k-bit precision needs to be output in the second iteration. In this way, the computation bit width of the next iteration always increases linearly based on the previous iteration, resulting in a significant increase of the area and latency. Based on the Babylonian algorithm, this paper proposes a method that successfully separates the total computation width of SRQT into the high and low parts. When the low bits are computed, the high bit width of the computation result does not need to be output with costs, which lowers the latency and reduces the area. On this basis, this paper further proposes a novel ultra-low-latency DP SQRT algorithm in combination with the Poly-K low-latency fast approximation capability and the theory of accurate rounding of indirect errors. The following sections will describe the algorithm in detail.

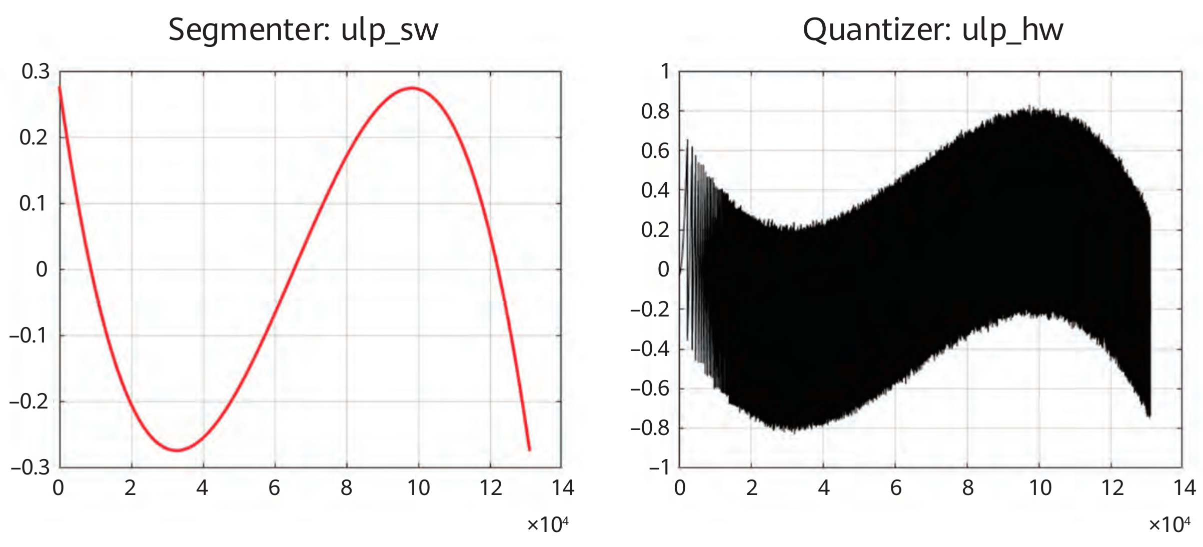


Figure 1 Error curves of the Poly-2 segmenter and quantizer

3.1 Computation Principle

In this section, x and $\tilde{x}$ are used to represent the floating-point number form and real number (a number with infinite precision in the mathematical sense) form of the same number, respectively. Unless otherwise specified, addition, subtraction, multiplication, division, and SQRT computation represented by $+, -, \times, \div, /, \sqrt{}$ are operations in the sense of real numbers.

3.1.1 Babylonian Algorithm Variant

$$\begin{cases} x = \tilde{f}^2 \in [1,4) \\ \tilde{f}^2 = (\tilde{f}_u + \tilde{f}_l)^2 \end{cases}$$

$$\tilde{f}_l = \frac{1}{\tilde{f}_u} \times \frac{x-\tilde{f}_u^2}{2} - \frac{\tilde{f}_l^2}{2\tilde{f}_u} \quad (1)$$

$$\tilde{f}_l \approx \frac{1}{\tilde{f}_u} \times \frac{x-\tilde{f}_u^2}{2} \quad (2)$$

As shown in the preceding formulas, the variant of the Babylonian algorithm splits the exact SQRT value $\tilde{f}$ into two parts: high-bit $\tilde{f}_u$ and low-bit $\tilde{f}_l$. When $\tilde{f}_u$ is known, the result of the low-bit $\tilde{f}_l$ can be approximately computed by using formula (2). With reference to the polynomial approximation theory, the foregoing process can be implemented in three steps:

1 Use piecewise polynomial approximation (Poly-K) to compute $\tilde{f}_u \approx \sqrt{x}$. The computation result is represented by floating-point number f_u.

2 Use Poly-K to compute $R = \frac{1}{\sqrt{x}} \approx \frac{1}{\tilde{f}_u}$.

3 Use formula (2) to compute $\tilde{f}_l$. The computation result is represented by floating-point number f_l.

Steps 1 and 2 can be performed in parallel.

3.1.2 Error Analysis

The difficulty of the SQRT algorithm that adopts quadratic convergence lies in error control to meet the rounding requirements of IEEE 754. This paper parametrically constructs an error propagation model and parses the constraints on the parameters that are required to achieve the target precision. The following part describes it in detail.

1 Truncation of input variable x

The initial input variable is used to compute f_u because f_u is only part of the exact SQRT value. Theoretically, only the higher bits of the mantissa of the input variable $x \in [1,2)$ need to be retained. Assume that $m+2$ fractional bits are taken to form x_t, and then Poly-K is used for computation. When the exponent of the input normal data of the FP SQRT or the exponent after denormal data is changed to normal data is an odd number, the mantissa needs to be multiplied by 2 before SQRT computation. In this case, only $m+1$ fractional bits are retained for $2x$. Therefore, when Poly-K computes f_u and R in the case of $x = x_t$ and $x = 2x_t$, there are two target functions and two tables internally. When the functions and tables are merged, the following formula can be obtained:

$$x = x_t + x_r \in [1,4), x_r \in [0, 2^{-(m+1)})$$

2 f_u computation

The computation error of f_u generated by the foregoing truncation operation is:

$$\widetilde{et} = \sqrt{x} - \sqrt{x_t} = \frac{x_r}{\sqrt{x}+\sqrt{x_t}} \in [0, 2^{-(m+2)})$$

When Poly-K is used to compute f_u, $m+1$ fractional bits are output, with a precision of $1\ ulp$:

$$\tilde{f}_u = f_u = SQRT_{[1,4)}^{Poly}(x_t)$$

$$\varepsilon 1 = \sqrt{x_t} - f_u, \qquad |\varepsilon 1| < 2^{-(m+1)} \quad (3)$$

Therefore, the total error of f_u, that is,
$\tilde{f}_l = \sqrt{x} - f_u = \sqrt{x} - \tilde{f}_u$, can be expressed as:

$$\begin{aligned} |\tilde{f}_l| &= |\sqrt{x} - f_u| = |(\sqrt{x} - \sqrt{x_t}) + (\sqrt{x_t} - f_u)| \\ &= |\tilde{e}t + \varepsilon 1| < 2^{-(m+2)} + 2^{-(m+1)} = \frac{3}{2} \cdot 2^{-(m+1)} \end{aligned} \quad (4)$$

3 R computation

When Poly-K is used to compute R, $m+1$ fractional bits are output, with a precision of $1\ ulp$:

$$R = RSQRT_{[1,4)}^{Poly}(x_t)$$

$$\varepsilon 2 = \frac{1}{\sqrt{x_t}} - R, \qquad |\varepsilon 2| < 2^{-(m+1)} \quad (5)$$

Based on formulas (3) and (5), we have:

$$\begin{cases} f_u = \sqrt{x_t} - \varepsilon 1 \\ R = \frac{1}{\sqrt{x_t}} - \varepsilon 2 \end{cases}$$

Further, the error $\tilde{\gamma} = \frac{1}{f_u} - R$ of R approximating $\frac{1}{f_u}$ can be expressed as follows (identity (1) is an expression about f_u):

$$\begin{aligned} |\tilde{\gamma}| &= \left|\frac{1}{f_u} - R\right| = \left|\frac{1}{\sqrt{x_t}-\varepsilon 1} - \frac{1}{\sqrt{x_t}} + \varepsilon 2\right| \\ &= \left|\frac{\varepsilon 1}{(\sqrt{x_t}-\varepsilon 1)\sqrt{x_t}} + \varepsilon 2\right| = \left|\frac{\varepsilon 1}{f_u\sqrt{x_t}} + \varepsilon 2\right| \end{aligned}$$

Because $1 \leq f_u, \sqrt{x_t} < 2$, we have:

$$|\tilde{\gamma}| = \left|\frac{1}{f_u} - R\right| \leq |\varepsilon 1| + |\varepsilon 2| < 2^{-(m+1)} + 2^{-(m+1)} = 2^{-m} \quad (6)$$

4 Computation of $\alpha = \frac{x - f_u^2}{2}$

In this step, no rounding operation is performed, and therefore no error will be generated. According to formula (4), $\sqrt{x} = f_u + \tilde{f}_l$, and $1 \leq f_u, \sqrt{x_t} < 2$, the value range of $\alpha = \frac{x - f_u^2}{2}$ can be determined as follows:

$$|\alpha| = \left|\frac{x - f_u^2}{2}\right| = \left|\frac{\tilde{f}_l^2 + 2f_u\tilde{f}_l}{2}\right| = \left|\tilde{f}_l \cdot \left(\frac{\sqrt{x}}{2} + \frac{f_u}{2}\right)\right| < \frac{3}{2} \cdot 2^{-m} \quad (7)$$

5 Discarded term $\tilde{\beta} = \frac{\tilde{f}_l^2}{2f_u}$

With reference to formula (4) and $f_u \in [1, 2)$, the discarded term introduces the following error:

$$\tilde{\beta} = \frac{f_l^2}{2f_u} \in \left[0, \frac{9}{8} \cdot 2^{-(2m+2)}\right) \quad (8)$$

6 Total error

By substituting formulas (6) to (8) into formula (1), we have:

$$\tilde{f}_l = \alpha(R + \tilde{\gamma}) - \tilde{\beta} = \alpha R + \left(\alpha\tilde{\gamma} - \tilde{\beta}\right) = \alpha R + \tilde{e} \quad (9)$$

where

$$\tilde{e} = \alpha\tilde{\gamma} - \tilde{\beta}, |\tilde{e}| = \left|\alpha\tilde{\gamma} - \tilde{\beta}\right| \leq |\alpha\tilde{\gamma}| + \left|\tilde{\beta}\right| < \frac{3}{2} \cdot 2^{-2m} + \frac{9}{8} \cdot 2^{-(2m+2)}$$

During $\sqrt{x}$ computation, N fractional bits are retained in the output f. According to formula (9), $f_l \approx \alpha R$ and N fractional bits need to be retained for f_l. To reduce errors, round half (RH) is used for multiplication operations, which will generate an error ($\tilde{e}_{RH}$) of $\frac{1}{2}$ *ulp* ($2^{-(N+1)}$). Then formula (9) can be refined as follows:

$$\tilde{f}_l = \alpha R + \tilde{e} = RH(\alpha R) + \tilde{e}_{RH} + \tilde{e}$$

Therefore, the total error $err = f_u + f_l - \sqrt{x} = f_l - \tilde{f}_l$ is:

$$\left|f_l - \tilde{f}_l\right| = |e + \tilde{e}_{RH}| < \frac{3}{2} \cdot 2^{-2m} + \frac{9}{8} \cdot 2^{-(2m+2)} + 2^{-(N+1)} \quad (10)$$

3.1.3 Error Control

We expect to control the error represented by formula (10) to stay within 1 *ulp*, that is,

$$|err| < 2^{-N} \quad (11)$$

According to formula (10), only the following condition must be met:

$$\frac{3}{2} \cdot 2^{-2m} + \frac{9}{8} \cdot 2^{-(2m+2)} < 2^{-(N+1)}$$

Let

$$m = \frac{N}{2} + \delta \quad (12)$$

Only the following condition must be met:

$$\frac{3}{2} \cdot 2^{-2\delta} + \frac{9}{8} \cdot 2^{-(2\delta+2)} \leq 2^{-1}$$

That is,

$$\delta > \frac{1}{2}\log_2\left(\frac{57}{16}\right) \approx 0.9164$$

In formula (12), if we let $\delta = 1$, that is,

$$m = \frac{N}{2} + 1 \quad (13)$$

Then formula (11) is true. When Poly-K is used to compute f_u and R, the input contains $m + 2 = \frac{N}{2} + 3$ fractional bits and the computation result contains $m + 1 = \frac{N}{2} + 2$ fractional bits. The Poly-K's computation error is within 1 *ulp* ($2^{-(\frac{N}{2}+2)}$).

By substituting formula (13) into formula (10), we have:

$$\begin{aligned}|err| &< \tfrac{3}{2} \cdot 2^{-(N+2)} + \tfrac{9}{8} \cdot 2^{-(N+4)} + 2^{-(N+1)} \\ &= 2^{-N} \cdot \left(\tfrac{1}{2} + \tfrac{3}{8} + \tfrac{9}{128}\right) = 2^{-N} \cdot \tfrac{121}{128} < 2^{-N}\end{aligned}$$

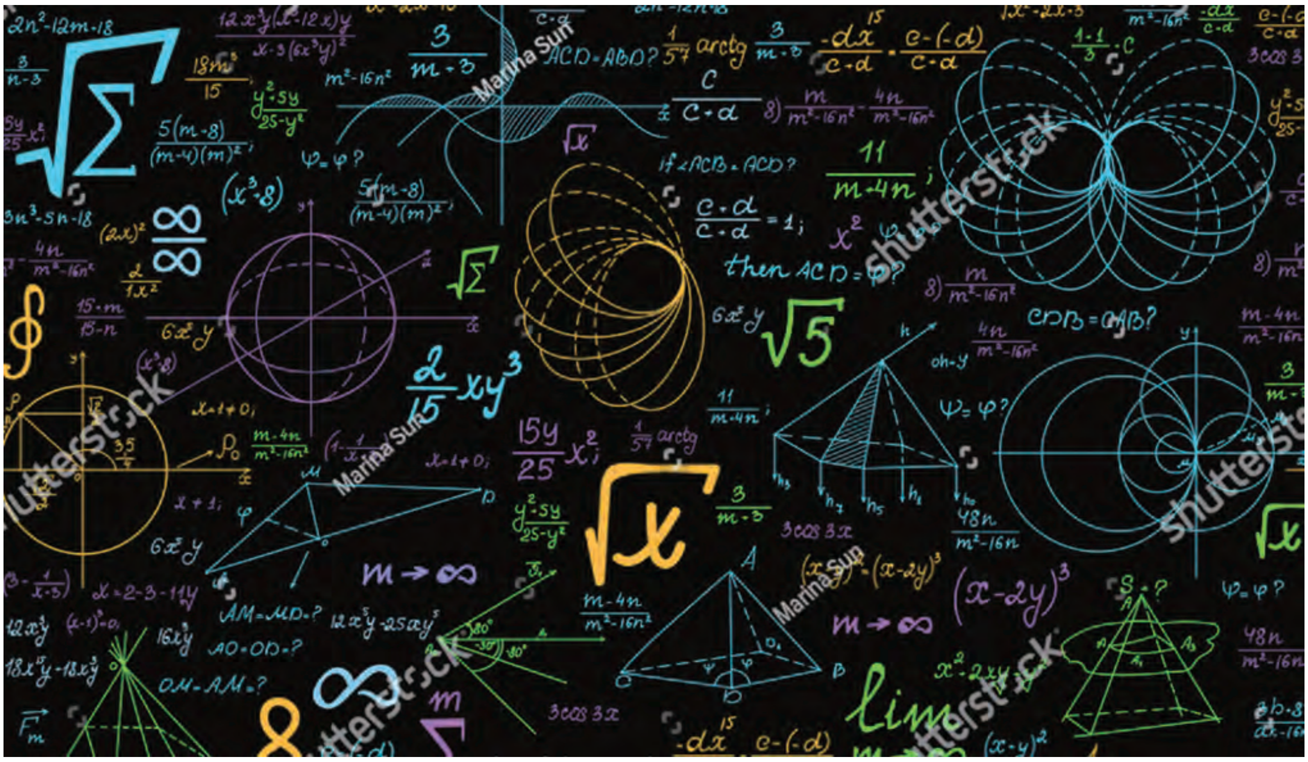
3.1.4 Bit Width Analysis

In this section, the bit width expression is used. That is, A.(B) indicates the data format with A-bit integers and B-bit decimals. For example, 2.(B) indicates 2-bit integers and B-bit decimals. To avoid confusion, all real decimals are represented in a fractional form or a negative exponent form, for example, $\frac{3}{4}$ or 2^{-1}.

1 Poly-K: f_u

Input: decimal part of the x_t mantissa, including $m + 2 = \frac{N}{2} + 3$ bits and with a value range of $[0, 1)$.

Output: $1.(m + 1) = 1.\left(\frac{N}{2} + 2\right)$, i.e. $\frac{N}{2} + 3$, with a value range of $[1, 2)$.



2 Poly-K: $R = \frac{1}{\sqrt{x_t}} \approx \frac{1}{f_u}$

Input: decimal part of the x_t mantissa, including $m+2 = \frac{N}{2}+3$ bits and with a value range of $[0, 1)$.

Output: $1.(m+1) = 1.\left(\frac{N}{2}+2\right)$, i.e. $\frac{N}{2}+3$, with a value range of [2^{-1}, 1]. (Due to the Poly-K approximation, the left interval is enlarged to a closed interval.)

3 Computation of $\alpha = \frac{x - f_u^2}{2}$

Input: If the processed exponent is an odd number, it is equivalent to multiplying x with a $1.N$ bit width by a coefficient 2. The resulting x bit width is $2.(N–1)$. Taking both even and odd exponents into consideration, the bit width of x should be $2.N$.

$$\alpha = \frac{x-f_u^2}{2} = \frac{2.N-1.(\frac{N}{2}+2)\times 1.(\frac{N}{2}+2)}{2} = \frac{2.N-2.(N+4)}{2}$$
$$= 1.(N+1) - 1.(N+5) = 1.(N+5) \quad (14)$$

Output: Formula (7) shows that $|\alpha| < 2^{-(m-1)} = 2^{-\frac{N}{2}}$, i.e. the significant bits of α start from the $\frac{N}{2}+1$-th fractional bit. Considering the sign bit, all fractional bits of the computation result need to be retained from the $\frac{N}{2}$-th fractional bit. With reference to formula (14), it can be learned that a total of $N+5-\left(\frac{N}{2}-1\right) = \frac{N}{2}+6$ least significant bits (LSBs) need to be retained.

In conclusion, during circuit computation, the following results can be obtained:

A x_c: The lower $\frac{N}{2}+2$ bits of $2.N$-bit x are used.

B $(f_u^2)_c$: $1.\left(\frac{N}{2}+2\right) \times 1.\left(\frac{N}{2}+2\right) -\!\!\rightarrow 2.(N+4) -\!\!\rightarrow \frac{N}{2}+6.$

During multiplication operations, only the lower $\frac{N}{2}+6$ bits of the result are output to compute α.

C $\alpha = x_c - (f_u^2)_c$:

$$\left(\frac{N}{2}+2\right) - \left\{1.\left(\frac{N}{2}+2\right) \times 1.\left(\frac{N}{2}+2\right) -\!\!\rightarrow \left(\frac{N}{2}+6\right)\right\} -\!\!\rightarrow \frac{N}{2}+6 \quad (15)$$

Note that the subtraction in formula (15) requires high-bit alignment.

4 $f_l \approx RH(\alpha R)$ computation: $\left(\frac{N}{2}+6\right) \times \left(\frac{N}{2}+3\right) -\!\!\rightarrow \frac{N}{2}$

During normal computation:

$$\alpha R = 1.(N+5) \times 1.\left(\frac{N}{2}+2\right) = 1.\left(\frac{3}{2}N+7\right) \xrightarrow{\text{Round Half}} 1.N \quad (16)$$

After RH is incorporated into the partial products matrix (PPM), the lower $\left(\frac{N}{2}+7\right)$ bits need to be truncated during multiplication.

According to formulas (4) and (11), the original value range of f_l is $|f_l| < \frac{3}{2} \cdot 2^{-\left(\frac{N}{2}+2\right)}$, and a jitter within the range of $(-2^{-N}, 2^{-N})$ is generated if $RH(\alpha R)$ is directly used for approximation computation. Therefore, the value range of $RH(\alpha R)$ is:

$$|f_l| < \frac{3}{2} \cdot 2^{-\left(\frac{N}{2}+2\right)} + 2^{-N}$$

Letting $N > 6$, the value range of $RH(\alpha R)$ can be scaled to:

$$|f_l| < 2^{-\left(\frac{N}{2}+1\right)}$$

It can be learned that the significant bits of $RH(\alpha R)$ start from the $\frac{N}{2}+2$-th fractional bit. Considering the sign bit, the significant bits need to be retained from the $\frac{N}{2}+1$-th bit.

In conclusion, the output of formula (16) has a total of $\frac{N}{2}$ bits, ranging from the $\frac{N}{2}+1$-th bit to the N-th bit after the decimal point.

According to formula (15), α in formula (16) can be represented by $\frac{N}{2}+6$ bits, being retained from the $\frac{N}{2}$-th fractional bit. Multiplying formula (16) by $2^{\frac{N}{2}}$, we have:

$$\alpha - \rightarrow \alpha' : 1.(N+5) - \rightarrow 1.\left(\frac{N}{2}+5\right)$$

$$RH(\alpha R) - \rightarrow RH(\alpha' R) : 1.N - \rightarrow 0.\frac{N}{2}$$

At the circuit implementation level, formula (16) can be rewritten as follows:

$$\alpha' R = 1.\left(\frac{N}{2}+5\right) \times 1.\left(\frac{N}{2}+2\right) = 0.(N+7) \xrightarrow{Round\ Half} 0.\frac{N}{2} \quad (17)$$

That is, $\left(\frac{N}{2}+7\right)$ decimals are truncated and $\frac{N}{2}$ decimals are retained. Therefore, the bit width expression of the operation is as follows:

$$\left(\frac{N}{2}+6\right) \times \left(\frac{N}{2}+3\right) - \rightarrow \frac{N}{2}$$

5 Computation of $f \approx f_u + f_l$

$$1.\left(\frac{N}{2}+2\right) + \left(0.\frac{N}{2}\right) \times 2^{-\frac{N}{2}} - \rightarrow 2.N$$

In the preceding formula, the first two bits of $N/2$-bit f_l must be aligned with the last two bits of f_u.

3.2 Accurate Rounding

If the mantissa of a floating-point number is $1.N$ and the exponent is an odd number, the bit width changes to $2.(N–1)$. After both the even and odd exponents are considered, the bit width of x changes to $2.N$, and the value range changes to $x = [1, 4)$. In this case, we have:

$$f^2 = (f_u + f_l)^2 \approx x \quad (18)$$

When FP SQRT computation complies with the rounding standard of IEEE 754, the special case of Tie To Even or Tie To Away will not occur in RH. Therefore, these two rounding modes can be combined into a simple RH, which will be proved in the next section. In addition, the output of FP SQRT computation must be non-negative, so To Zero and To Negative can be combined into a simple round down (RD). In conclusion, FP SQRT supports three rounding modes: round up (RU), RH, and RD.

Since we control the error of formula (18) within $1ulp = 2^{-N}$, i.e. $\left|f - \sqrt{x}\right| < 2^{-N}$, formula $(f+ulp)^2 > x > (f-ulp)^2$ is true. The exact rounding value F must be selected from:

$$\begin{cases} f = f_u + f_l \\ f_p = f + ulp \\ f_n = f - ulp \end{cases}$$

3.2.1 Rounding Logic

Assume that the accurate SQRT computation result is $f_b = \sqrt{x}$ and the indirect error $ie0$ is defined as follows:

$$ie0 = f^2 - x \quad (19)$$

$$2.N \times 2.N - 2.N = 3.2N - 2.N - \rightarrow 4.2N$$

1 When $ie0 == 0$, F in three rounding modes is selected from f.

2 When $ie0 > 0$, $f > f_b > f_n$ and F is the one closer to f_b in $\{f, f_n\}$:

$$(f - f_b) - (f_b - f_n) = (f + f_n) - 2f_b = (2f - ulp) - 2f_b$$

Because only the positive and negative properties of the preceding formula are concerned and $f > ulp$, the positive and negative properties of $(2f - ulp) - 2f_b$ are equivalent to those of $ie0n$.

$$\begin{aligned} ie0n &= \left[(2f - ulp)^2 - 4f_b^2\right]/4 \\ &= \left(4f^2 - 4ulp \times f + ulp^2 - 4x\right)/4 = ie0 - ulp \times f + ulp^2/4 \end{aligned}$$

$$4.2N - 2^{-N} \times 2.N + 2^{-(2N+2)} = 4.2N - 0.2N + 2^{-(2N+2)}$$
$$= 4.2N + 2^{-(2N+2)}$$

It can be learned from the preceding formulas that the value of $ie0n$ is equivalent to the value of $4.2N$ plus 2'b01. Therefore, the value of $ie0n$ cannot be 0, indicating that the special case of Tie To Even or Tie To Away will not occur. As such, we can omit the computation of additional item $ulp^2/4$:

$$ie0n = ie0 - ulp \times f \tag{20}$$

Table 1 SQRT accurate rounding logic

RM / F	RU	RH	RD
f_p	$ie0 < 0$	$iep0 < 0\#$	/
f	$ie0 \geq 0$	else	$ie0 \leq 0$
f_n	/	$ie0n \geq 0*$	$ie0 > 0$

*: $ie0n = ie0 + ien \geq 0 - \rightarrow ie0 \geq -ien$. When the error is within 1 ulp, $ien < 0$ is always true. Therefore, $ie0n \geq 0$ contains $ie0 > 0$.

#: $iep0 = iep + ie0 < 0 - \rightarrow ie0 < -iep$. When the error is within 1 ulp, $iep > 0$ is always true. Therefore, $iep0 < 0$ contains $ie0 < 0$.

3 When $ie0 < 0$, $f < f_b < f_p$. F is the one closer to f_b in (f, f_p):

$$(f_p - f_b) - (f_b - f) = (f_p + f) - 2f_b = (2f + ulp) - 2f_b$$

Similarly, the positive and negative properties of $(2f + ulp) - 2f_b$ are equivalent to those of $iep0$.

$$iep0 = \left[(2f + ulp)^2 - 4f_b^2\right]/4$$
$$= \left(4f^2 + 4ulp \times f + ulp^2 - 4x\right)/4 = ie0 + ulp \times f + ulp^2/4$$

$$4.2N + 2^{-N} \times 2.N + 2^{-(2N+2)}$$
$$= 4.2N + 0.2N + 2^{-(2N+2)} = 4.2N + 2^{-(2N+2)}$$

It can be learned from the preceding formulas that the value of $iep0$ is equivalent to the value of $4.2N$ plus 2'b01. Therefore, the value of $iep0$ cannot be 0, indicating that the special case of Tie To Even or Tie To Away will not occur. As such, we can omit the computation of additional item $ulp^2/4$:

$$iep0 = ie0 + ulp \times f \tag{21}$$

The positive and negative properties of formulas (20) and (21) can express the error relationships of the options $\{f_p, f, f_n\}$. Considering the additional item, $ie0n$ or $ie0p$ will not be 0. However, after the additional item is omitted, $ie0n$ and $ie0p$ can be 0. Table 1 represents the FP SQRT accurate rounding logic.

The options $\{f_p, f, f_n\}$ are mutually exclusive. Therefore, the selection logic described in the preceding table can be further simplified. Assume that the selection signal of F is a 2-bit FS:

$$FS = \{SP, SN\} \tag{22}$$

where SP indicates selecting f_p, and SN indicates selecting f_n. They can be computed as follows:

$$SP = (RU\&(ie0 < 0))|(RU\&(iep0 < 0)) \tag{23}$$

$$SN = (RH\&(ie0n \geq 0))|(RD\&(ie0 > 0)) \tag{24}$$

In this case, we have:

$$F = \begin{cases} f_p, & ifFS = 2'b10 \\ f_n, & ifFS = 2'b01 \\ f, & ifFS = else \end{cases} \tag{25}$$

In conclusion, the rounding logic computation can be represented by formulas (19)–(21), and the rounding logic selection can be represented by formulas (22)–(25).

3.2.2 Computing Optimization and Bit Width Analysis

1 $ie0$

For formula (19), the value range of the computation result is analyzed as follows:

$$ie0 = f^2 - x = \left(f - \sqrt{x}\right)\left(f + \sqrt{x}\right) = \left(-2^{-N}, 2^{-N}\right) \times \left(2\sqrt{x} + \left(-2^{-N}, 2^{-N}\right)\right)$$
$$= \left(-2^{-N}, 2^{-N}\right) \times \left(\left[2, 2\sqrt{2\left(2 - 2^{-N}\right)}\right] + \left(-2^{-N}, 2^{-N}\right)\right)$$
$$= \left(-2^{-N}, 2^{-N}\right) \times \left(2 - 2^{-N}, 2\sqrt{2\left(2 - 2^{-N}\right)} + 2^{-N}\right)$$
$$= \left(-2^{-N}, 2^{-N}\right) \times \left(2\sqrt{2\left(2 - 2^{-N}\right)} + 2^{-N}\right) \in \left(-2^{-N}, 2^{-N}\right) \times 4$$
$$= \left(-2^{-(N-2)}, 2^{-(N-2)}\right)$$

It can be learned that the significant bits of $ie0$ start from the $N-1$-th fractional bit. Considering the sign bit, the significant bits of $ie0$ need to be retained from the $N-2$-th fractional bit. In normal computation, the bit width expression of formula (19) is as follows:

$$ie0 = f^2 - x = 2.N \times 2.N - 2.N - \rightarrow 3.(2N) - 2.N = 4.(2N) \tag{26}$$

Therefore, the significant bits of $ie0$ must be retained to the $2N$-th fractional bit, with a total of $N+3$ bits. Similarly, the significant bits of all intermediate computation results of formula (26) only need to be retained to the $2N$-th fractional bit from the $N-2$-th fractional bit, with a total of $N+3$ bits.

The bit width for the multiplication operation of formula (26) is reduced below. Since

$$ie0 = f^2 - x = (f_u + f_l)^2 - x = f_u^2 + f_l^2 + 2f_u f_l - x \quad (27)$$

It can be learned from formulas (14) and (15) that $f_u^2 = 1.\left(\frac{N}{2}+2\right) \times 1.\left(\frac{N}{2}+2\right) = 2.\left(N+4\right)$, where the lower $\frac{N}{2}+6$ bits have been computed. The last seven bits (from the $N-2$-th fractional bit to the $N+4$-th fractional bit) are taken for computation in formula (27).

In formula (27), f_l^2 and $2f_u f_l$ need to be computed. Based on formulas (16) and (17), we have:

$$f_l^2 = \left(0.\left(\frac{N}{2}\right) \times 2^{-\frac{N}{2}}\right)^2 = 0.\frac{N}{2} \times 0.\frac{N}{2} \times 2^{-N} = 0.N \times 2^{-N} \quad (28)$$

$$\frac{N}{2} \times \frac{N}{2} - \rightarrow N$$

That is, the computation result of f_l^2 has N bits (from the $N+1$-th fractional bit to the $2N$-th fractional bit) and is completely within the valid output range of formula (27). Therefore, all bits of the computation result are retained. Similarly

$$\begin{aligned} 2f_u f_l &= 2 \times 1.\left(\tfrac{N}{2}+2\right)\left(0.\tfrac{N}{2} \times 2^{-\frac{N}{2}}\right) \\ &= 2 \times 1.\left(N+2\right) \times 2^{-\frac{N}{2}} = 2.\left(N+1\right) \times 2^{-\frac{N}{2}} \end{aligned} \quad (29)$$

That is, the significant bits of $2f_u f_l$ start from the $\frac{N}{2}-1$-th fractional bit and ends with the $\frac{3N}{2}+1$-th fractional bit. Based on the output range of formula (27), the computation result of formula (29) must be retained to the $\frac{3N}{2}+1$-th fractional bit from the $N-2$-th fractional bit, with a total of $\frac{N}{2}+4$ bits.

To sum up, circuit computation in formula (27) involves:

A f_u^2: takes the last seven bits from the computed $\frac{N}{2}+6$ bits, starting from the $N-2$-th fractional bit to the $N+4$-th fractional bit.

B f_l^2: $\frac{N}{2} \times \frac{N}{2} - \rightarrow N$, starting from the $N+1$-th fractional bit to the $2N$-th fractional bit.

C $2f_u f_l$: $\left(\frac{N}{2}+3\right) \times \frac{N}{2} - \rightarrow \frac{N}{2} + 4(LSB)$, starting from the $N-2$-th fractional bit to the $\frac{3N}{2}+1$-th fractional bit.

D x: $2.N - \rightarrow 3(LSB)$, starting from the $N-2$-th fractional bit to the N-th fractional bit.

Among the preceding four items, only B and C need to be computed, and the sum of A and D can be computed in advance. Finally, the sum of the four items is:

$$7 + N + \left(\frac{N}{2}+4\right) - 3 - \rightarrow N+3 \quad (30)$$

2 $ie0n$ and $ie0p$

We perform the following shift operation on the computation result of formula (19):

$$ie0s = ie0 \times 2^N = 3.N \quad (31)$$

Moving the computation results of formulas (20) and (21) to the left by N bits and combining them with formula (31), we have:

$$ie0ns = ie0n \times 2^N = 2^N \times ie0 - 2^N \times ulp \times f = ie0s - f \quad (32)$$

$3.N - 2.N - \rightarrow 4.N$, where only the most significant sign bit is taken.

$$iep0s = iep0 \times 2^N = 2^N \times ie0 + 2^N \times ulp \times f = ie0s + f \quad (33)$$

$3.N + 2.N - \rightarrow 4.N$, where only the most significant sign bit is taken.

$ie0ns$ and $iep0s$ can be computed by using formulas (32) and (33). So far, the derivation of the accurate rounding computation model of the FP SQRT is completed based on formulas (31)–(33) and accurate rounding expressions (22)–(25).

3.3 Summary of the General Model

For FP SQRT computation whose output contains N fractional bits, m is configured according to formula (13), f_u and R are computed by using Poly-K, and a computation process of the Babylonian algorithm variant is completed according to formula (14)–(17) to obtain f_l. The entire FP SQRT computation process has an error of $1\ ulp$. Finally, according to formula (31)–(33), the accurate rounding process of FP SQRT is completed.

In this paper, the software compiles the foregoing general computation model, sets the parameter N to a value from

$\{30, 32, 34, 36, 38, 40\}$, and traverses all the input and output results to verify the model correctness. When $N = 52$, the software compiles the corresponding RTL Pilot Code and passes the smoke test.

In the general model, if $N = 52$ and $m = 27$, the DP SQRT can be computed.

4 Ultra-Low Latency Microarchitecture Design

In this paper, Poly-K is utilized to compute the SQRT f_u that contains 28 fractional bits (high bits of the DP SQRT) and the reciprocal SQRT R. Because the output precision is high, the piecewise second-order approximation, that is, Poly-2, is used. The Babylonian algorithm variant is then used to compute the 26-bit f_l (low bits of the DP SQRT). Finally, the indirect error theory is used to achieve accurate rounding. For hardware design, this paper fully considers the timing question. For some multiplication and addition operations, the carry-save format is directly used to perform booth encoding [29, 30], rather than performing addition on the carry-save format before booth encoding, to reduce the latency. Figure 2 shows an ultra-low latency microarchitecture design corresponding to the DP SQRT algorithm proposed in this paper.

Some expressions are defined as follows:

- 0p17: indicates the bit width, consisting of 0-bit integers and 17-bit decimals.

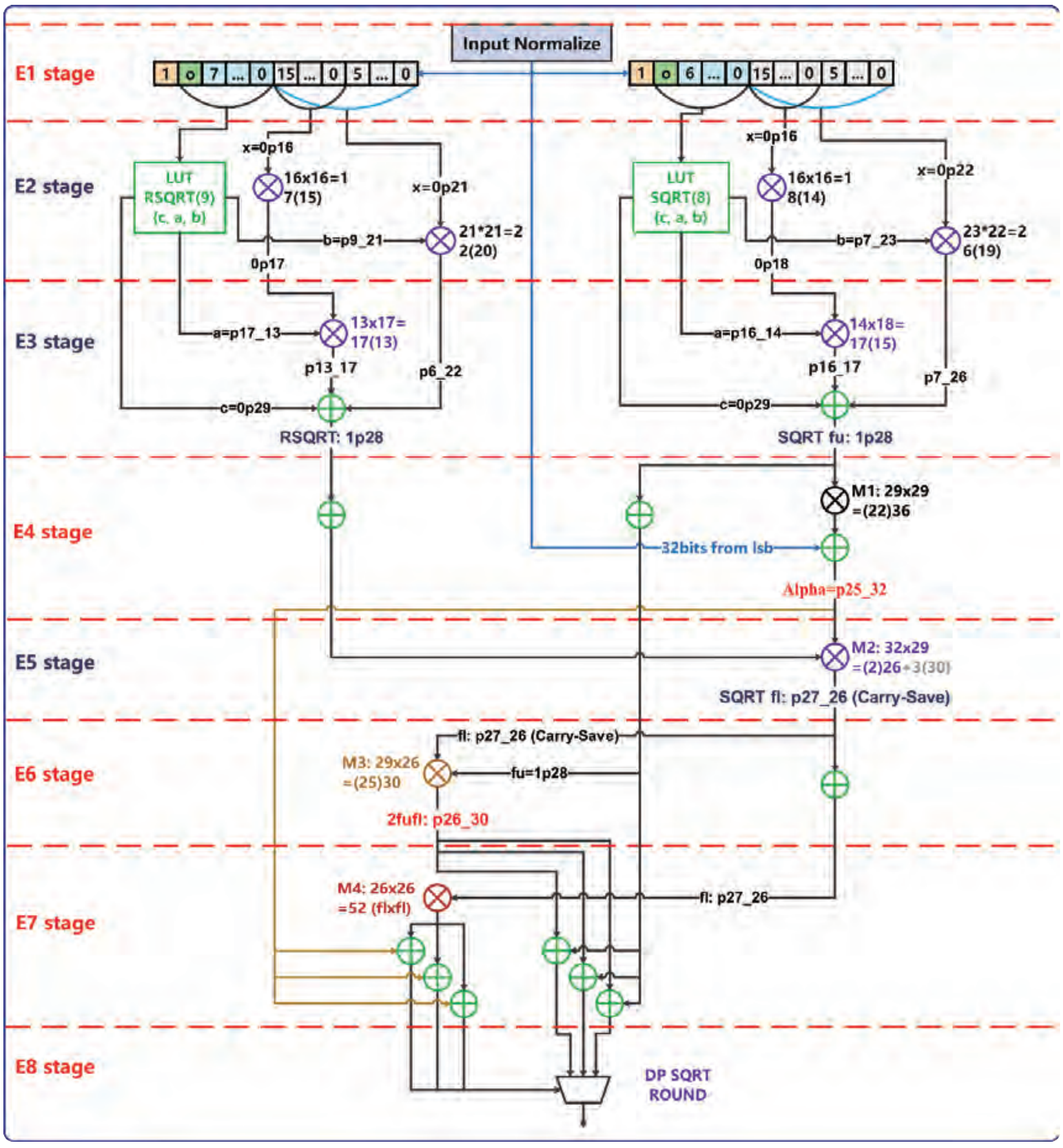
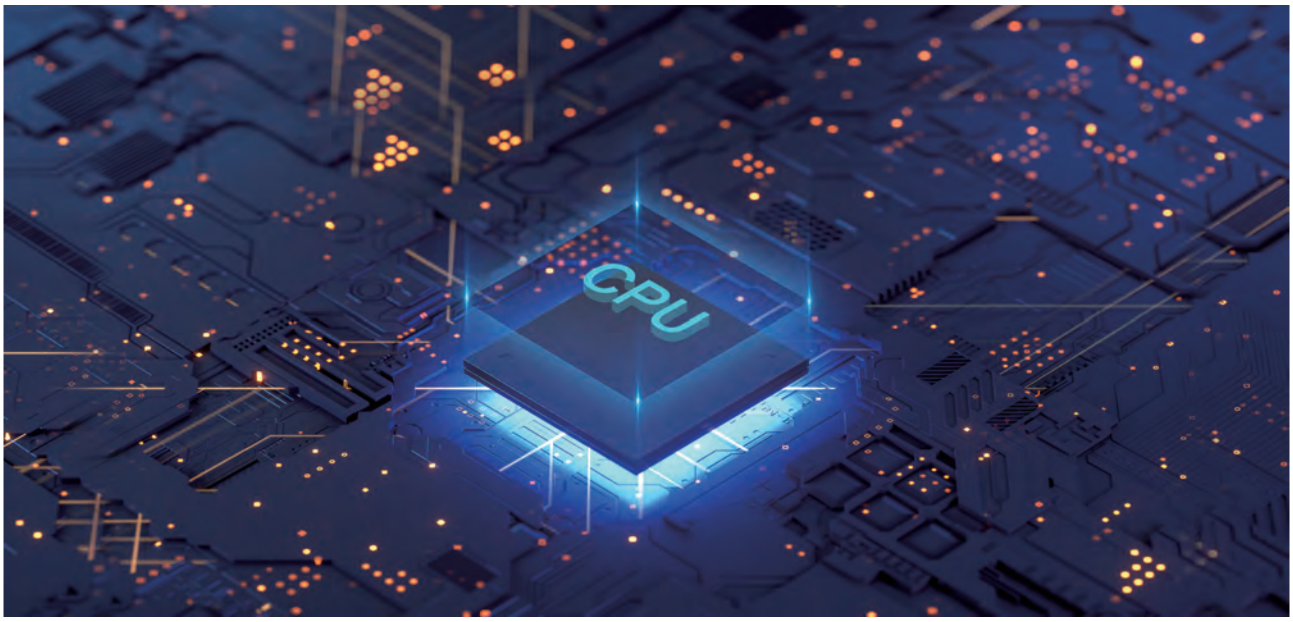


Figure 2 DP SQRT microarchitecture



- p14_13: indicates the position of significant bits, that is, a total of 13 significant bits are retained, starting from the 14th fractional bit.
- a, b, and c: represent the quadratic term coefficient, monomial term coefficient, and constant term coefficient of Poly-2, respectively.

This design uses eight cycles to complete the DP SQRT computation for a 52-bit mantissa, and supports pipeline operations. Specifically, E1 stage formats floating-point numbers, E2 and E3 stages solve the SQRT that contains 28 fractional bits and the reciprocal SQRT, E4 stage computes Alpha (α), E5 stage computes the low bits of DP SQRT, and E6–E8 stages complete the accurate rounding process.

In Figure 2, the two multipliers at E6 and E7 stages can be completed in parallel if the DP SQRT is independently implemented. In this way, a latency of one cycle can be reduced. In practice, this design is realized together with the double-precision floating-point divider (DP DIV) designed by the authors by means of resource reuse. In such a scenario, parallel-to-serial conversion is performed, and the reuse logic can reduce the cost of one multiplier. Therefore, this paper maintains the serial idea. The following parts will describe the DP SQRT microarchitecture design in detail.

4.1 Floating-Point Data Normalization: E1 Stage

After the input floating-point data is normalized, the mantissa is represented in the following form based on the parity feature of the exponent.

$$\begin{cases} \mathbf{1.}\,xxx\cdots xxx,\ exp = even \\ \mathbf{1x.}\,xxx\cdots xx,\ exp = odd \end{cases}$$

The parity flag bit and the 7 bits (for SRQT table lookup) and 8 bits (for reciprocal SRQT table lookup) after the mantissa integer 1 are combined to form the coefficient table lookup input for Poly-2 computation.

4.2 Poly-2 SQRT and Reciprocal Computation: E2 and E3 Stages

4.2.1 Poly-2: SQRT

E2: A 16×16 multiplier generates 32 bits, and the higher 18 bits are taken as x^2. A 23×22 multiplier generates 45 bits, and the higher 26 bits are taken as bx.

E3: Based on x^2, a 14×18 multiplier generates 32 bits, and the higher 17 bits are taken as ax^2. ax^2, bx, and c are summed up to obtain a 1.28-bit result. The result is represented in the carry-save format, with booth encoding completed (four XOR logic depths estimated).

4.2.2 Poly-2: RSQRT

E2: A 16×16 multiplier generates 32 bits, and the higher 17 bits are taken as x^2. A 21×21 multiplier generates 42 bits, and the higher 22 bits are taken as bx.

E3: Based on x^2, a 13×17 multiplier generates 30 bits, and the higher 17 bits are taken as ax^2. Then ax^2, bx, and c are summed up to obtain a 1.28-bit result. The result is represented in the carry-save format, with booth encoding completed (four XOR logic depths estimated).

4.3 Alpha Computation: E4 Stage

The following three computations need to be performed in parallel at this stage:

1 A 29-bit adder completes summation for RSQRT with carry-save encoding and outputs a 1.28-bit result.

2 A 29-bit adder completes summation for SQRT with carry-save encoding and outputs a 1.28-bit result.

3 A square booth multiplier, combined with a 32-bit partial product, computes Alpha (32 bits) in the carry-save format.

4.4 SRQT Low-Bit Computation: E5 Stage

A 32×29 booth multiplier is used to generate 26+3 bits in the carry-save format, and booth encoding is performed to obtain a result in the f_l format.

4.5 Accurate Rounding: E6–E8 Stages

At E6 stage, the following two computations are performed in parallel:

1 A 26×29 booth multiplier computes $2f_u f_l$ and outputs a result in the carry-save format.

2 A 26-bit adder completes summation for f_l in the carry-save format.

At E7 stage, three remainders and three SRQT options are computed in parallel.

1 One 26×26 booth multiplier and three adders are used to compute $ie0s$, $ie0ns$, and $ie0ps$, and generate the SP (f_p selected), SN (f_n selected), and S (f selected) logic. This process is not shown in Figure 2.

2 Three adders, combined with constants $+1$ and -1, are used to compute f, f_p, and f_n.

At E8 stage, f, f_p, and f_n are selected based on $\{SP, SN, S\}$.

5 Performance Comparison

Table 2 Performance comparison between the very-high-radix SRT and proposal

	Very High Radix (2^{12})	Proposal
Latency (Cycle)	15	8
Throughput (Samples per Cycle)	0.1	1

In table 2, the latency and throughput of the current advanced very-high-radix SRT implementation [6] is compared with those of the proposal and microarchitecture implementation. In the SRT microarchitecture design, pre-processing and post-processing require five cycles to iterate in time division multiplexing (TDM) mode. Each iteration requires two-cycle cost of latency and outputs 12 bits. Therefore, it takes 15 cycles for the SRT algorithm to compute the DP SQRT. The ultra-low-latency DP SQRT implementation proposed in this paper has almost half the latency of the original implementation. In addition, due to its algorithm characteristics and pipeline design, the throughput can reach up to 1, which is 10 times that of the SRT algorithm. In high-performance CPU scalar units and vector/SIMD applications, high throughput and low latency are urgent requirements, substantiating the value of our research.

6 Summary

This paper proposes a novel DP SQRT algorithm and microarchitecture design based on polynomial fast approximation, separation of high and low bits, and accurate rounding of indirect errors. Compared with the SRT algorithm and hardware implementation used in the academia and industry, the DP SQRT algorithm and microarchitecture feature high throughput and ultra-low latency. The design idea can be used not only for DP SQRT computation, but also for high-precision complex function computation such as DP DIV and INT64 DIV, representing good scalability. This provides a chance for a new breakthrough for the long-standing, complex ALU research.

References

[1] Parhami, Behrooz. Computer arithmetic. Vol. 20. No. 00. Oxford university press, 2010.

[2] Majerski, Stanislaw. "Square-root algorithms for high-speed digital circuits." IEEE 6th Symposium on Computer Arithmetic (ARITH). IEEE, 1983.

[3] Lang, Tomas, and Paolo Montuschi. "Higher radix square root with prescaling." IEEE Transactions on Computers 41.08 (1992): 996–1009.

[4] Oberman, Stuart F., and Michael J. Flynn. "Design issues in division and other floating-point operations." IEEE Transactions on computers 46.2 (1997): 154–161.

[5] Lang, Tomas, and Paolo Montuschi. "Very-high radix combined division and square root with prescaling and selection by rounding." Proceedings of the 12th Symposium on Computer Arithmetic. IEEE, 1995.

[6] Lang, Tomás, and Paolo Montuschi. "Very high radix square root with prescaling and rounding and a combined division/square root unit." IEEE Transactions on Computers 48.8 (1999): 827–841.

[7] Ercegovac, Miloš D., and Jean-Michel Muller. "Complex square root with operand prescaling." The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology 49.1 (2007): 19–30.

[8] Wang, Dong, and Miloš D. Ercegovac. "A radix-16 combined complex division/square root unit with operand prescaling." IEEE Transactions on Computers 61.9 (2011): 1243–1255.

[9] Freiman, C. V. "Statistical analysis of certain binary division algorithms." Proceedings of the IRE 49.1 (1961): 91–103.

[10] Harris, D., S. Oberman, and M. Horowitz. "SRT division: Architectures, models, and implementations." Computer Systems Library, Standard University, Tech. Rep (1998).

[11] Higham, Nicholas J. "Newton's method for the matrix square root." Mathematics of computation 46.174 (1986): 537–549.

[12] King, Richard F., and David L. Phillips. "The logarithmic error and Newton's method for the square root." Communications of the ACM 12.2 (1969): 87–88.

[13] Ercegovac, Milos D., *et al.* "Improving Goldschmidt division, square root, and square root reciprocal." IEEE Transactions on Computers 49.7 (2000): 759–763.

[14] Pineiro, J-A., and Javier D. Bruguera. "High-speed double-precision computation of reciprocal, division, square root, and inverse square root." IEEE Transactions on Computers 51.12 (2002): 1377–1388.

[15] Fowler, David, and Eleanor Robson. "Square root approximations in Old Babylonian mathematics: YBC 7289 in context." Historia Mathematica 25.4 (1998): 366–378.

[16] Kosheleva, Olga. "Babylonian method of computing the square root: Justifications based on fuzzy techniques and on computational complexity." NAFIPS 2009–2009 Annual Meeting of the North American Fuzzy Information Processing Society. IEEE, 2009.

[17] Zuras, Dan, *et al.* "IEEE standard for floating-point arithmetic." IEEE Std 754.2008 (2008): 1–70.

[18] Strollo, Antonio GM, Davide De Caro, and Nicola Petra. "Elementary functions hardware implementation using constrained piecewise-polynomial approximations." IEEE Transactions on Computers 60.3 (2010): 418–432.

[19] De Caro, Davide, Nicola Petra, and Antonio GM Strollo. "High-performance special function unit for programmable 3-D graphics processors." IEEE Transactions on Circuits and Systems I: Regular Papers 56.9 (2008): 1968–1978.

[20] De Caro, Davide, *et al.* "Minimizing coefficients wordlength for piecewise-polynomial hardware function evaluation with exact or faithful rounding." IEEE Transactions on Circuits and Systems I: Regular Papers 64.5 (2017): 1187–1200.

[21] Lee, Dong-U., *et al.* "Hardware implementation trade-offs of polynomial approximations and interpolations." IEEE Transactions on computers 57.5 (2008): 686–701.

[22] Hsiao, Shen-Fu, *et al.* "Design of hardware function evaluators using low-overhead nonuniform

segmentation with address remapping." IEEE transactions on very large scale integration (VLSI) systems 21.5 (2012): 875–886.

[23] Sun, Huaqing, *et al.* "A universal method of linear approximation with controllable error for the efficient implementation of transcendental functions." IEEE Transactions on Circuits and Systems I: Regular Papers 67.1 (2019): 177–188.

[24] Dong, Hongxi, *et al.* "PLAC: Piecewise linear approximation computation for all nonlinear unary functions." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 28.9 (2020): 2014–2027.

[25] Muller, Jean-Michel, and Jean-Michael Muller. Elementary functions. Birkhũser Boston, 2006.

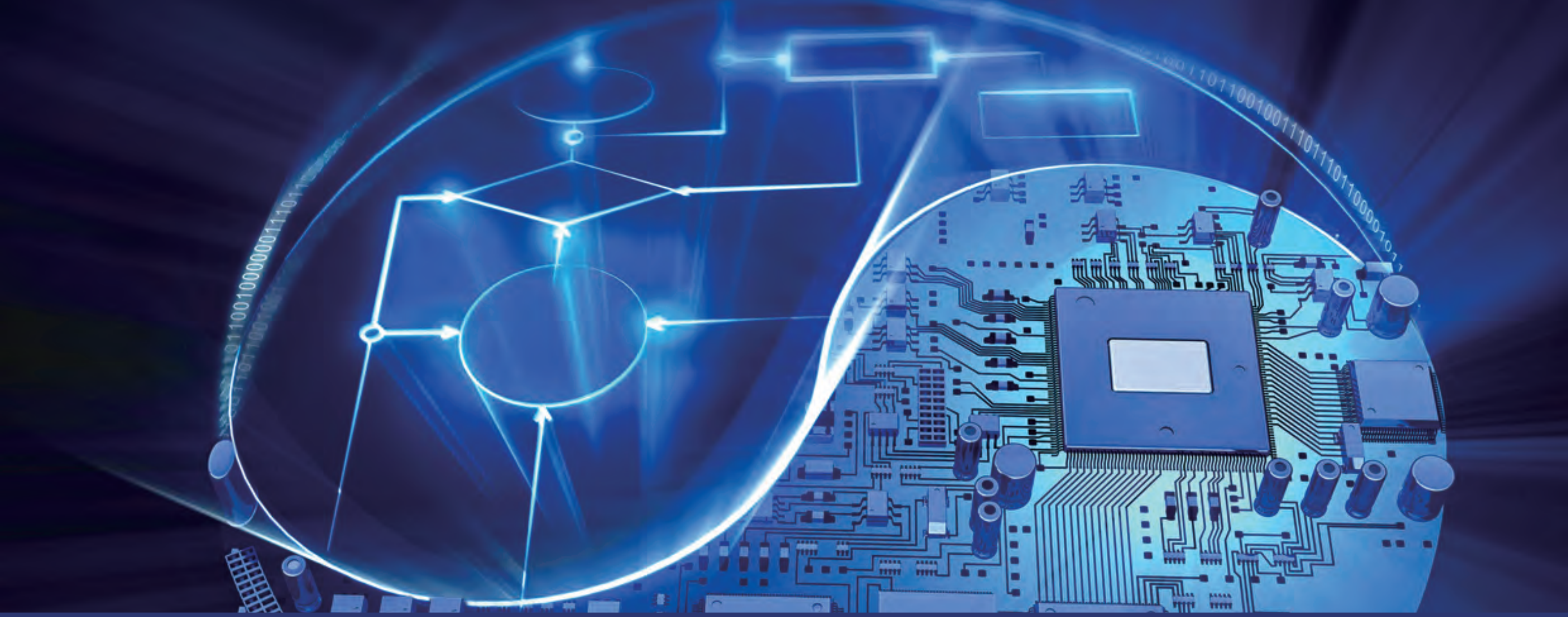
[26] Jeraj, Janez, and V. John Mathews. "Identification of nonlinear, memoryless systems using Chebyshev nodes." Proceedings. (ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005, Vol. 4. IEEE, 2005.

[27] Pachón, Ricardo, and Lloyd N. Trefethen. "Barycentric-Remez algorithms for best polynomial approximation in the chebfun system." BIT Numerical Mathematics 49.4 (2009): 721–741.

[28] Arzelier, Denis, Florent Bréhard, and Mioara Joldes. "Exchange algorithm for evaluation and approximation error-optimized polynomials." 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH). IEEE, 2019.

[29] Tsoumanis, Kostas, *et al.* "An optimized modified booth recoder for efficient design of the add-multiply operator." IEEE Transactions on Circuits and Systems I: Regular Papers 61.4 (2014): 1133–1143.

[30] Aravind, A. R., *et al.* "Study on modified booth recoder with fused add-multiply operator." AIP Conference Proceedings. Vol. 2393. No. 1. AIP Publishing LLC, 2022.



Balancing the Yin and Yang of Dynamic Compilation and Execution

Opportunities for Hardware–Software Co-Design

Maria Carpen-Amarie, Rene Mueller, Konstantinos Tovletoglou *

Abstract

Modern language implementations leverage dynamic or just-in-time (JIT) compilation and take advantage of their unique opportunity to observe the state of a program as it is being executed. By collecting dynamic information about control flow, types, and values, the compiler is able to apply speculative optimizations when generating machine code. The more is known about the program's execution, the more aggressive the speculation can be for achieving a higher performance gain. However, speculations come at a cost. Profiling information about the program's state must be collected through instrumentation, which constitutes a run-time overhead. Furthermore, non-provable assumptions made by the compiler that enable speculations must be checked at run-time and a deoptimization mechanism must be provided if those speculations fail, further increasing the run-time overhead. As a result, current JIT compilers are very risk-averse. A reduction of both overheads would allow collecting more accurate profiles while also enabling more risk-seeking speculations in the compiler.

In this paper, we assert that language runtime implementations offer an interesting avenue for hardware-software co-design to build better dynamic compilers and processors that execute dynamically compiled programs. While observing the increasing importance of high-productivity languages, we ask how one would build a hardware-assisted execution platform for them. While we acknowledge the value of better informed hardware, we also argue for better informed software. Software-directed narrowly focused profiling in hardware — either by using novel profiling units or by leveraging the existing performance monitoring units — provides profiles that are more detailed at low overhead. We also propose hardware-assisted deoptimization and assertions with user-level trap handlers to further lower the checks and deoptimization costs. We outline interesting areas in the Yin and Yang between JIT compilation and the execution, peak performance and overhead, and better hardware and JIT software for dynamically compiled languages.

Keywords

AOT, JIT, runtime environment

* Authors are listed in alphabetical order

1 Introduction

In dynamic compilation, the translation of programs is delayed until execution. Dynamic compilation is usually performed by a just-in-time (JIT) compiler while some functions of the program are already being executed. As a result, the JIT compiler can leverage dynamic program information that is not available to a static compiler, which produces executables ahead-of-time (AOT). Dynamic compilation is used for managed statically typed languages, such as Java and C#, and even more importantly, for dynamically typed languages (e.g., JavaScript). Run-time optimizations performed by JIT compilers are essential for executing programs written in dynamically typed languages efficiently. For example, modern JIT compilers in JavaScript engines, such as V8, JavaScriptCore, and SpiderMonkey, generate specialized machine code based on the data types encountered during execution.

The ability of the compiler to obtain information about the program's execution provides interesting optimization opportunities. Unlike in traditional profile-guided optimization, this ability eliminates the need to choose a representative workload from which the profile information is captured. Instead, profile information is collected during the execution of the actual workload. The flip side of JIT compilation is that the profile information must be gathered before the program is translated to native code, increasing the start-up time and the overhead at the beginning of the execution. Nevertheless, we believe that the dynamic compilation overheads can be further reduced through a hardware-software co-design and, if combined with caching and AOT techniques, ultimately produce a superior execution performance over traditional static compilation.

In this article, we shed light on the space between compilation and execution, as well as between optimization and profiling. We discuss improvements to current JIT compilers and potential hardware extensions that enable better JITs and more efficient program execution.

1.1 JITs Make Dynamic Languages Faster

JIT-based execution of dynamically typed languages offers a significant performance advantage over interpreted execution. In this section, we use a Python example to demonstrate such performance gains. Specifically, we implement a textbook version of the multiplication of two dense matrices in idiomatic Python, using only core features of the language itself without the help of any library. Because Python does not support arrays in its core language, let alone multi-dimensional arrays, we represent matrices with Python `list` objects. We provide two implementations. The first represents a matrix as a nested list (a list of lists, requiring five lines of Python code), which permits the use of the common index operator syntax `mat[i][j]`. In the second implementation, using seven lines of Python code, we flatten each matrix to a single list in a row-major format, such that element accesses are of the form `mat[i * NUM_COLS + j]`.

We measure the performance of the two implementations in CPython, which is the reference implementation of the Python language. Even though CPython only contains an interpreter[1] — the bytecode instructions are executed inside a large `switch/case` statement — it is the most widely used runtime today. We compare the CPython interpreter with two JIT-based Python runtimes, PyPy [1] and GraalPython [2]. For reference, we also compare against two native implementations: NumPy, a parallel and highly optimized library for numerical computing in Python, and a C++ implementation of the textbook algorithm exposed to Python. Both native variants use NumPy's array format and therefore cannot be considered a strictly idiomatic use of Python.

Figure 1 shows the different execution times for the multiplication of two 128×128 matrices. As can be seen from the figure, the JIT-based executions are about two orders of magnitude faster than CPython, with the flattened list implementation being faster. The key observation is that the JIT-based execution of the Python version is approximately on-par with the same implementation in C++. This means that a native implementation in C++ does not provide a performance advantage over regular Python in a JIT-based runtime. As such, we can achieve good performance for the textbook algorithm without changing the language and without C/C++ foreign-function binding to Python. For comparison, we also show the performance of a highly optimized implementation in NumPy based on the `dgemm` routine from the Intel Math Kernel Library (MKL) that leverages thread- and SIMD-parallelism. This implementation offers more than another order-of-magnitude speedup on a 16-core system, but it is far from what could be considered a textbook implementation.

[1] At the time of writing, there are discussions to add a type of JIT-based execution to CPython 3.12 by specializing bytecode instructions for types (PEP 659).

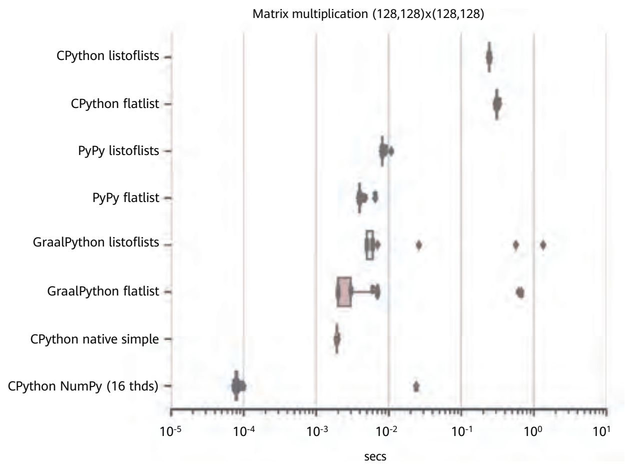


Figure 1 Matrix multiplication: textbook Python implementations in CPython, PyPy, and GraalPython. We also show the same implementation in C++/pybind11 and the optimized version from NumPy (parallel, tuned SIMD) for comparison.

Of note is the fact that all the pure-Python implementations are single-threaded. In order to maintain compatibility with CPython's single-thread runtime and its global interpreter lock, there is no auto-parallelization in either of the Python JITs.

We are not suggesting that the use of optimized libraries such as NumPy be discontinued. Rather, we are making two important points: (1) an efficient JIT-enabled language runtime may provide sufficient performance such that the programmer does not need to fall back on an implementation in C++, and (2) Python itself is not a slow language. The example illustrates that JIT-based runtimes allow the use of high-productivity languages without sacrificing performance. In case of single-threaded execution, the performance of JIT-compiled Python is comparable with that of C++ native code. In this paper, we want to push JIT compilers further, addressing the aforementioned overheads from profiling and run-time checks from speculation. We provide a brief background on JIT compilation in the next section, and then propose techniques to build better JITs with the help of new hardware functionality.

2 Just-in-Time Compilation

There are different approaches to JIT compilation: *trace-based JITs*, which compile code from application traces across functions; *region-based JITs*, which compile coarser regions of code; and *method-based JITs*, which operate on individual methods and functions at a time. PyPy, used in the example provided in the previous section, is a tracing JIT for Python. More precisely, it is a meta-tracing JIT, in which the JIT traces the Python interpreter interpreting the Python program. Tracing JITs are widely used in *dynamic binary optimization*, *translation*, and *instrumentation* [3, 4]. In this paper, we focus on method-based JITs because they are used in the HotSpot Java Virtual Machine (JVM) and V8 JavaScript engine.

2.1 A Focus on Method-Based JITs

Practical JIT compilers typically require a trade-off between compilation effort and quality of the generated code in order to reduce the application start-up times. Spending more compilation effort on parts of a program that are frequently executed (also called "hot"), such as inner loops of methods, is more beneficial than spending it on other parts that are rarely executed. A baseline compiler provides sufficient performance for infrequently executed parts, whereas compilers that are more sophisticated are used for critical parts. Managed runtimes therefore do not usually consist of only one single JIT compiler. Instead, they contain multiple so-called *compiler tiers*, with each tier spending a different amount of compilation effort. When methods or loops have been executed more frequently, they are compiled by higher tiers. In other words, the code *tiers up* through the different compilers as it gets more frequently executed. Furthermore, program execution often begins in an interpreter. This has the advantage that instructions executed rarely (e.g., class initialization) do not need to be compiled at all.

In the process of tiering up, the language runtime collects information about the program. Such information includes the number of times a method was called, how often a then- and else-branch was taken, the number of iterations of unbounded loops, counts for each case clause in a `switch/case` statement, or the actual receiver types in virtual method calls. This information is collected by the interpreter and in the compiled code through instrumentation. The information is then made available for optimization in the higher compiler tiers.

Figure 2 shows the four compiler tiers of the HotSpot JVM. The tiers are provided by two different compilers, called *C1* and *C2*. C1 is the fast compiler that generates code quickly, whereas C2 is the best-optimizing compiler. While Tiers 1 to 3 are served by the same C1 compiler, they differ in the amount of code instrumentation applied for profiling. Tier 1 does not apply any profiling, whereas code generated by Tier 3 is fully instrumented and therefore has the highest

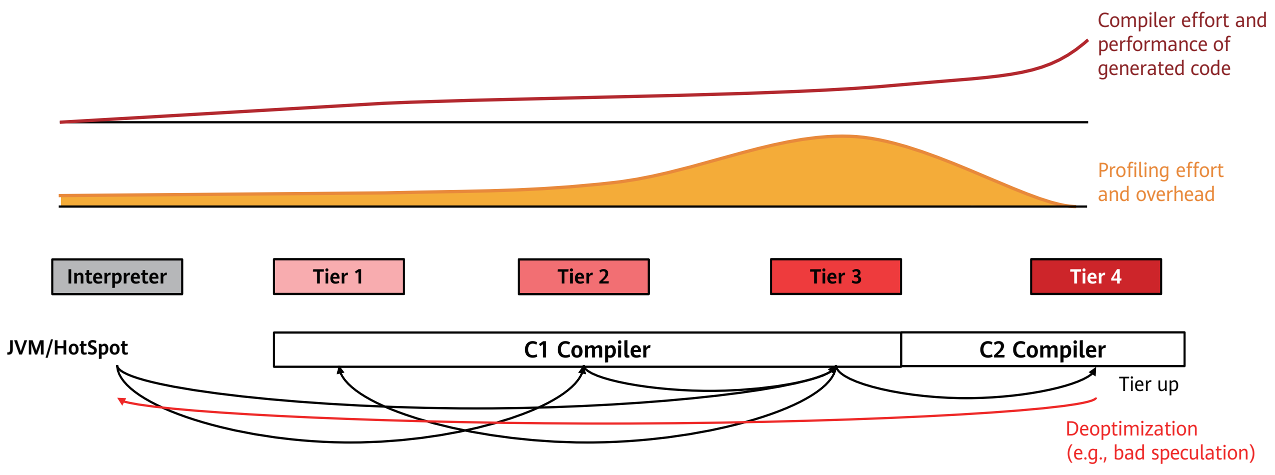

Compiler effort and performance of generated code
Profiling effort and overhead
Interpreter
Tier 1
Tier 2
Tier 3
Tier 4
JVM/HotSpot
C1 Compiler
C2 Compiler
Tier up
Deoptimization (e.g., bad speculation)

Figure 2 JIT-based language runtimes often consist of multiple JIT compilers with varying compilation efforts. As code gets "hotter," it tiers up to higher compiler tiers that spend more compilation effort but produce better code. If speculations made by the compilers fail at run-time, the execution of the code stops and it is resumed at a lower tier (deoptimization).

profiling overhead. The profiling data collected by executing Tier 3 code is used by the C2 compiler if the code transitions to the highest compiler tier. Note that tiering does not necessarily follow a strict order from Tier 1 to 4. Code starting from the interpreter usually moves directly to Tier 3 (compilation with full profiling enabled) and then to Tier 4 (full optimizing compiler). There are circumstances in which code follows a different order. For example, simple code — such as in getter methods — takes the path to Tiers 3 → 1. This is because there is little useful information gained from profiling such simple methods, and these methods will likely be inlined soon anyway when compiling the calling methods. Tier 2 is generally used if the queues that hold compilation tasks for the C2 compiler threads become full. The JVM then applies back-pressure and moves code from Tier 3 to a lower Tier 2 until the pressure on the compiler threads eases up.

The highest compiler tier spends the highest optimization effort. It uses all available profiling data collected to produce the best performing code. Typically, the highest compiler tiers do not emit instructions for application profiling due to the overheads, which in this last phase of the tiering-up process would eliminate most of the optimization gains.

In method-based JITs, the code tiers up not only at the granularity of functions (methods) but also at the level of individual loops. Typically, loops get "hot" before methods. After the loop is compiled, execution resumes in the compiled code. Before the loop can be executed, the current interpreter stack frame needs to be converted into the native machine representation. This is called *On-Stack Replacement (OSR).*

The Tier 4 compiler also applies speculative optimizations; specifically, it applies optimizations on branches not taken or on encountered dynamic types. This generally simplifies the code and reduces its size but requires additional instructions — *guard checks* — that assert that the speculative assumptions made by the compiler still hold. If an assumption no longer holds, the execution of the generated code needs to stop, the generated code needs to be discarded, and the execution needs to be resumed in a lower tier (e.g., in the interpreter). This step is called *deoptimization* or *OSR Exit*. Because this step requires translating the machine thread state (the call stack with the machine registers) into the abstract program state of the virtual machine, this operation is expensive.

2.2 No Universal JIT: Implementation Matters

From an application standpoint, not all JIT environments provide the same performance and optimizations out of the box. Given three different runtime implementations from the same family of languages (i.e., HotSpot, Graal Community Edition (CE), and OpenJ9 Java VMs), one would expect comparable performance when executing the same benchmark suite several times with the same options for the runtime and benchmarks. However, the results are quite

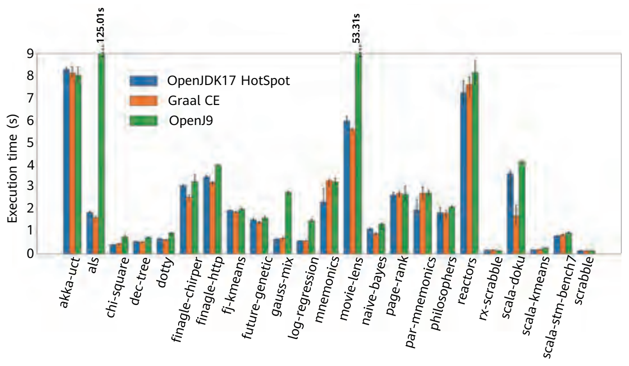


Figure 3 Renaissance benchmarks' performance (wall-clock time) with three different JVMs, median/min/max over 11 runs with default options

different from one JVM to another, as illustrated in Figure 3 on the Renaissance benchmark suite [5]. The figure shows the wall-clock time per benchmark iteration.

We look at these performance numbers from the perspective of the average user, who sees the language runtime as a black-box and is unaware of any internal details. The first oddity is the very low performance of OpenJ9 for two of the workloads: `als` and `movie-lens`. Underneath, these two benchmarks implement the same algorithm, *alternating least squares*, and they both use a Spark engine.

Setting these two applications aside, OpenJ9 consistently shows worse performance than the other two runtimes. One could argue that a better-tailored configuration of OpenJ9 would provide increased performance. However, when the default parameters for JIT heuristics are poorly chosen, it is extremely difficult for the user to decide what would be a good selection, as there are numerous parameters that could be altered.

The HotSpot and Graal runtimes are similar in terms of performance, despite the maturity of the former and the novelty of the latter. Further parameter tuning for any of them would be as difficult as for OpenJ9. Reportedly, the Graal Enterprise Edition (EE) has better-tuned features than the CE version, but it is not open to the public.

With so much variability in related runtimes, the gap between different classes of languages is even larger. Dynamically typed languages need different optimizations than statically typed languages do. For example, while the V8 JavaScript engine focuses on type speculation, it makes little use of profiling, which is an important feature of the HotSpot compiler.

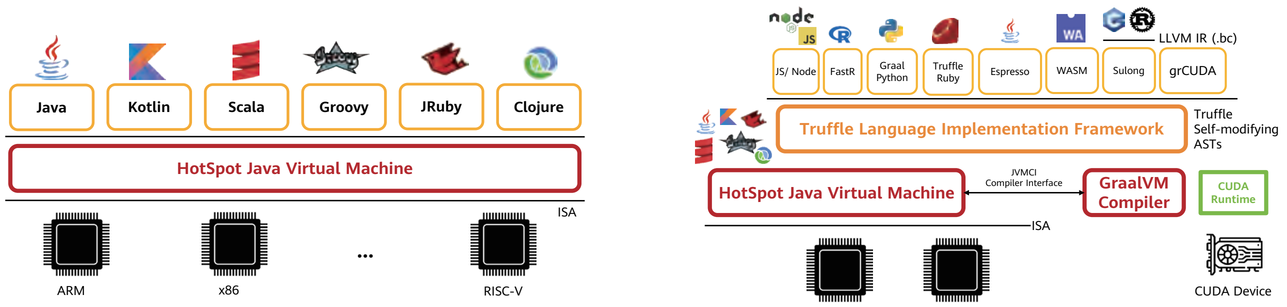
The takeaway here is that the performance of one JIT compiler does not allow any predictions to be made on the performance of another JIT compiler. Not all JITs are made equal. They not only strongly depend on whether the target language is statically or dynamically typed but also on a number of sensibly-chosen defaults and engineering subtleties. It is important to be aware of the specific characteristics of the runtime in order to avoid performance penalties.

2.3 JIT vs. DBO

Both JIT compilers and *dynamic binary optimization (DBO)* systems are a type of *dynamic optimizer*. While they are both capable of applying the same kind of optimizations to the code, the former operates on a *directly interpretable representation*, whereas the latter operates on a *directly executable representation* [6]. In other words, DBO systems need to lift the machine code into a type of intermediate representation (IR) in order to apply the required optimizations. Conversely, JIT-based platforms already operate on an IR, such as the Java bytecode for the JVM, and can directly employ it. Another difference between a JIT compiler and a DBO system is the degree of portability: DBOs are usually strictly tied to a specific architecture. Both JIT and DBO techniques use profiling (in software or in hardware) to identify optimization opportunities. As an interesting case for DBO, Zhou et al. [7] show how profile-guided optimization can be used for dynamic binary parallelization of single-threaded applications. They obtain up to 6× speedup on 8 threads over DynamoRIO [3].

A more generic version of a DBO is a *dynamic binary translator (DBT)*. While a DBO system is designed to do native-to-native binary translation and optimization in particular, DBT translates between different architectures, with or without optimizing the code in the process.

Another technique from the same family is feedback-driven optimization (FDO). Two large-scale studies from Google [8] and Facebook/Meta [9] give a sense of the gains obtained with FDO and post-link optimization in their respective data centers. More precisely, they perform fleetwide profiling of application binaries and deploy the new optimized code once it is generated. Google's FDO technique achieves a performance speedup of up to 10%, while Facebook/Meta's Bolt achieves up to 20% on top of FDO. However, because FDO-like systems are static optimizers, they miss out on many optimizations enabled by the dynamic nature of JIT or DBO.



(a) Java HotSpot runtime

(b) Truffle/GraalVM runtime with polyglot interop

Figure 4 Shared language runtimes

In this paper, we focus on JIT compilers for high-productivity dynamic languages, where the profiling and feedback loop are readily included in the compilation mechanism. Being an integral component of a language runtime, the JIT compiler naturally optimizes the code as it goes, relying on speculation and recovery to generate correct and efficient code.

3 Polyglot Language Runtimes

The implementations of efficient language runtimes are relatively complex, particularly if they are JIT-based. For example, the V8 JavaScript engine consists of about 1 million lines of code, while OpenJDK — the open-source implementation of the HotSpot JVM — consists of about 800 thousand lines. The implementation effort can be amortized by leveraging the runtime for multiple languages. This is possible if the language runtimes use the same intermediate representation. For the JVM, this is the Java bytecode. It is used as a compilation target for several high-level languages besides Java, such as Kotlin, Scala, Groovy, JRuby, and Clojure (see Figure 4a). A further advantage of a common language runtime is that it creates an ecosystem in which libraries can be shared across languages. For example, a Clojure program may use a library written in Java.

Rather than compiling to the common bytecode [10], language implementations can also share code of higher-level abstractions such as the abstract syntax tree. One notable example is Oracle's GraalVM with the Truffle language implementation framework [11] (see Figure 4b). While GraalVM is based on the HotSpot VM, it replaces the C2 compiler with its own Graal compiler. For this discussion, the relevant component is Truffle, which provides a framework that can be leveraged by language implementers. When using Truffle, these implementers can create interpreters using Truffle's abstract Syntax trees and, in turn, obtain an optimizing JIT compiler provided by Truffle/GraalVM for free. Truffle implementations for several languages already exist, such as for JavaScript, Python, R, and with Sulong [12], even C++ and Rust through LLVM bitcode.

Besides providing a JIT compilation for interpreters, Truffle/GraalVM is also a polyglot runtime that not only allows exchanging libraries but also provides interoperability between languages. Objects can be directly exchanged between programs written in different languages *without serialization*. An example is grCUDA [13]. This Truffle language for CUDA provides GPU-binding to all languages of the Truffle/Graal ecosystem.

We illustrate object sharing in an example between grCUDA and JavaScript (Listing 1). Existing GPU kernels, either in raw CUDA source code or compiled to binaries, can be brought into the Truffle ecosystem and bound as a callable Truffle object. Similarly, device arrays that hold data referenced by the GPU kernel can be allocated from any Truffle language (e.g., JavaScript) through interop with grCUDA. The resulting device array object is exposed to JavaScript as an ordinary JavaScript array. Zero-overhead access to this array, which is stored in Unified Memory, is provided through the JIT compiler. Once all the device arrays used by the GPU kernel are initialized by the JavaScript code, the kernel can be launched by simply invoking the callable object to which the kernel is bound and passing the references to the device arrays as arguments.

Listing 1 grCUDA: using a C++ kernel from JavaScript through polyglot language interop

```
// C++: CUDA kernel compiled into kernel.cubin
__global__ void incr(int *arr, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
     arr[i] = 1;
}

// JavaScript: bind and use kernel
const cu = Polyglot.eval('grcuda', 'CU')
const incKernel = cu.bindkernel('kernel.cubin',
   // kernel signature
  'cxx incr(arr: inout pointer sint32,n: sint32)'
)

// Allocate device array
const n = 100
const deviceArray = cu.DeviceArray('int', n)
for (let i = 0; i < numElements; i++) {
    // access like an ordinary JS array
    deviceArray[i] = i
}
// Launch kernel as 1 block with 128 threads
incKernel(1, 128)(deviceArray, n)
```

Truffle's ability to exchange objects even between dynamically typed languages comes from an additional abstraction besides the shared abstract syntax tree: objects are self-describing. The JIT compiler can query the objects about their properties (e.g., about the presence of certain members and whether they are indexable or callable) and speculatively generate machine code. In the grCUDA example in Listing 1, the compiler can map index accesses to loads and stores to the device array in Unified Memory. The legacy C foreign function interface that has traditionally been used between languages is no longer needed.

Because runtimes offer a vast range of optimization opportunities horizontally and vertically in the software stack, they are already regarded as central system components. And in cloud deployments, these components are increasingly important building blocks. We believe that they will gradually replace container and Micro VMs. A product already available in this direction is *Compute@Edge* by Fastly. It provides a WebAssembly runtime as a serverless environment onto which users can deploy microservices or functions written in any language that can be compiled to WebAssembly[2]. A possible use for shared runtimes is to co-locate functions or services from the same tenant, thereby allowing cross-layer optimization through the language runtime. Stronger isolation of different tenants is possible by encapsulating the language runtimes themselves into isolation containers, such as the Nabla Containers [14], in which the language runtime runs on top of a small Unikernel (Library OS).

[2] In order to use OS-like functionality, a WebAssembly System Interface (WASI) must be available and callable as a library from the respective languages.

4 Profiling

The JIT compiler requires profiling data about frequently executed parts of a program in order to decide which methods and loops to compile. The profiling data includes invocation counts, number of loop trips, frequencies of taken paths in branch instructions, and encountered types or values in critical places in the program. Subsequent compilations will leverage this information in order to generate machine code that is more efficient.

Today, this profiling information is primarily collected through code instrumentation. For example, the C1 JIT compiler of HotSpot produces machine code that gathers profiling information for each method and at each program location in the bytecode — specifically, at each *bytecode index (BCI)* — for a number of bytecode instructions. Table 1 lists the information collected per method per BCI. This information is written into the *Method's Data Object (MDO)* during the program's execution. The MDO is a vector that consists of one or more slots for each bytecode instruction for which profiling information is collected.

For methods, the generated code counts the number of invocations and deoptimizations, as well as the number of taken back-edges in the method (i.e., the total number of trips in a loop within the method). For branch instructions, the number of taken and fall-through paths is counted using two slots in the MDO. Similarly, for switch-case

Table 1 The HotSpot C1 JIT compiler collects specific profile data for each method and the following Java bytecode instructions.

Point of Instrumentation	Profile Data (Slots)
per-method data	Invocations, back-edges and decompilation count
if <condition>	Taken and not-taken count
goto	Taken count
tableswitch	Count for each case, default count
instanceof, checkcast, invokeinterface, invokevirtual	Up to two concrete receiver types and counts, count others, count non-profiled
invokestatic, invokespecial	Invocation count

instructions (such as `tableswitch`), the number of times each case was selected is counted in a separate slot. One slot is used for unconditional jump instructions (i.e., `goto`). Virtual method calls (i.e., `invokevirtual`) and type casts (i.e., `checkcast`) use five slots, recording the first two types encountered and their respective counts as key-count pairs. The fifth slot is used to count "other" types (i.e., neither the first nor the second type). The compiler uses this information to make inlining decisions. It distinguishes between monomorphic call-sites (in which only one type is encountered), bimorphic sites (with only two distinct types), and megamorphic sites. In addition, static invocations, such as static methods or constructors, are also counted.

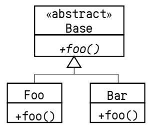
The profiling code generated by the C1 JIT compiler can be costly because it might be executed thousands of times before the method is identified as being sufficiently hot to be recompiled by the higher tier C2 compiler. In the next section, we quantify the overhead introduced by this profiling from two perspectives: (1) in terms of the extra instructions that are generated for profiling, and (2) in terms of performance overhead by comparing the execution times of the same Java workload, once with and once without profiling instrumentation.

4.1 Profiling Overheads

We demonstrate the increase of the code size from the added profiling instrumentation on a small example with a virtual method call. Figure 5a shows two classes — `Foo` and `Bar` — that both implement the method foo defined in the abstract class `Base`. A reference of the type `Base` can point to an instance of `Foo` or `Bar`. Now, consider a call to `foo` as shown in the test method illustrated in Figure 5b. The call to `foo` from the reference `obj[i]` is a virtual call because the implementation to be invoked is chosen at runtime based on the type of the referenced object: either `Foo::foo` or `Bar::foo`. When the C1 compiler in Tier 3 generates code for `invokevirtual` to `foo` at BCI 3, shown in the bytecode snippet in Figure 5c, it also emits code that profiles the receiver type of the call. The emitted code for the `invokevirtual` is shown in Figure 5d. The additional instructions used to collect the type profile are shown in red. These instructions load the address of the `test` method's MDO (from the address `0x7f9be3caa918`), compare the type (`klass`) of the `obj[i]` with the types stored in slots #1 and #2, and increment the respective counter if there is a match. If none of the slots match but one of the two slots has not yet been assigned to a type, this free slot is used, and its type and its counter are set. Otherwise, if none of the slots match and no free slot can be found, the generic slot "others" is incremented. Only after the code dedicated for profiling the type has been run is the virtual call made.

Of the 133 native instructions emitted for the method `test`, 35 of them are just for profiling purposes[3]. Besides increasing the size of the code footprint and the run-time overhead, the additional instructions can also disturb the micro-architectural state of the core, thrashing the cache hierarchy and putting more stress on other components, such as the branch predictors and address translation units.

In order to estimate the profiling overhead, we have devised an experiment in which we run the same Java workload once with profiling and once without. We allow or disallow profiling by limiting the tier to which HotSpot can tier up. Specifically, we only tier up to the C1 JIT compiler, namely, Tier 3 or Tier 1. In this way, if we stop tiering-up at Tier 1, there is no profiling code, but at Tier 3, the JIT compiler emits machine code that collects the full profile of the



(a) UML diagram

```java
public int test(Base[] obj, int i) {
  // virtual call
  return obj[i].foo();
}
```

(b) Java source code

```
0: aload 1
1: iload 2
2: aload
3: invokevirtual Base.foo
6: ireturn
```

(c) Java bytecode

```
mov     %rsi,%rdi           ; rdi <- rsi    (rdi <- this = receiver)
movabs  $0x7f9be3caa918,%rbx; metadata for invokevirtual 'foo' @BCI3
mov     0x8(%rdi),%edi      ; lookup klass of obj[i]
movabs  $0x800000000,%r10   ; decompress reference of target's klass
add     %r10,%rdi           ; decompress reference of target's klass
cmp     0x180(%rbx),%rdi    ; check if target's klass match with slot #1
jne     0x00007f9c21cb6fe3  ; not equal, try slot #2
addq    $0x1,0x188(%rbx)    ; increment count in type slot #1
jmpq    0x00007f9c21cb7049  ; proceed to vcall
cmp     0x190(%rbx),%rdi    ; check if target's klass match with slot #2
jne     0x00007f9c21cb6ff9  ; not equal, proceed to check if slot #1 is empty
addq    $0x1,0x198(%rbx)    ; increment count in type slot #2
jmpq    0x00007f9c21cb7049  ; proceed to vcall
cmpq    $0x0,0x180(%rbx)    ; target's klass didn't match any slot
                            ;   check if slot #1 is free
jne     0x00007f9c21cb701d  ; slot #1 is not free, try slot #2
mov     %rdi,0x180(%rbx)    ; store target's klass into the free slot #1
movq    $0x1,0x188(%rbx)    ; set slot #1 count to 1
jmpq    0x00007f9c21cb7049  ; proceed to vcall
cmpq    $0x0,0x190(%rbx)    ; check if slot #2 is free
jne     0x00007f9c21cb7041  ; slot #2 is not free, increment "other" slot
mov     %rdi,0x190(%rbx)    ; store target's klass into the free slot #2
movq    $0x1,0x198(%rbx)    ; set slot #2 count to 1
jmpq    0x00007f9c21cb7049  ; proceed to vcall
addq    $0x1,0x170(%rbx)    ; none of the slots matched and no free slot found
                            ;   increment count in "other" slot
movabs  $0x80101f610,%rax   ; make virtual call via Inline Cache
callq   0x7f9c215a9ac0      ; *invokevirtual foo
```

(d) Assembly code with profiling instrumentation

```
mov     %rsi,%rdi           ; rdi <- rsi    (rdi <- this = receiver)
prof_dist64  2, %rdi, $0x7f9be3caa918; count distinct %rdi using 2 slots
                                     ;   using metadata at $0x7f9be3caa918
movabs  $0x80101f610,%rax   ; make virtual call via Inline Cache
callq   0x7f9c215a9ac0      ; *invokevirtual foo
```

(e) Assembly code with the proposed profiling instruction

Figure 5 (a) UML diagram showing the definition of the abstract class `Base` and two concrete subclasses, `Foo` and `Bar`. (b) Java code of the method test, which calls the virtual method `foo`. (c) The Java bytecode generated for the method test. (d) Snippet of the machine code generated by HotSpot JVM — the code shown in red is responsible for profiling the virtual call. (e) Snippet of the machine code using the proposed ISA extension to profile and update the metadata of the virtual call.

[3] Figure 5d does not show the entire native code generated for the test method, nor does it show the profiling instructions that precede or follow the virtual call.

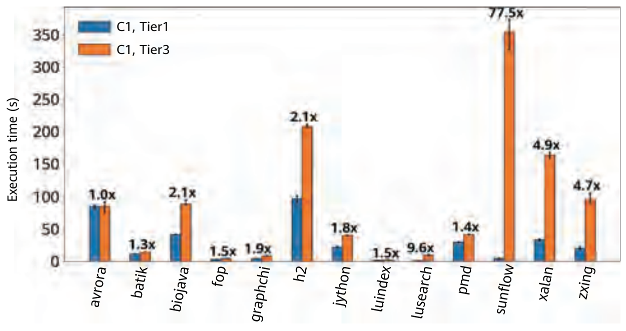


Figure 6 Execution time across 13 DaCapo benchmarks that are limited to using the C1 JIT compiler up-to Tier 1 and Tier 3, respectively. The difference in performance is due to the profiling code emitted in Tier 3 compared to Tier 1.

bytecode program. In both cases, the machine code is generated by the same compiler (C1) using the same level of optimizations. Figure 6 shows the execution times of 13 benchmarks from the DaCapo benchmark suite [15] for the code generated by the HotSpot C1 JIT compiler limited at Tier 1 and at Tier 3. The overhead of profiling can have a significant performance impact in many workloads, resulting in up to a 77× higher execution time. On average, the code with profiling (Tier 3) is 8.5× slower than without it (Tier 1). These numbers explain why the highest optimizing compiler, C2 in Tier 4, does not emit any profiling code anymore — doing so would simply be too expensive.

4.2 Current PMU-Based Profiling

As demonstrated in the previous section, collecting the profile data listed in Table 1 through instrumentation introduces a high run-time overhead. Ideally, the code profiling should be offloaded to dedicated hardware units that record the required data. In this section, we discuss the possibility of leveraging the performance monitoring units (PMUs) already existing in modern CPU cores to help collecting this information. In the next section, we will propose possible novel extensions to the PMUs to increase the coverage of the collected information, improve the accuracy, and further lower the run-time overheads.

OpenJ9 is the first JVM that takes advantage of hardware support for profiling. In OpenJ9, profiling combines both software- and hardware-based techniques. Similar to the HotSpot JVM, the OpenJ9 JVM performs code instrumentation in the highest profiling tier. However, code in the lower profiling compiler tiers can leverage hardware to identify hot methods. When OpenJ9 runs under Linux or AIX on an IBM POWER8 system, it leverages the CPU core's hardware capabilities [16, 17] for sample-based profiling

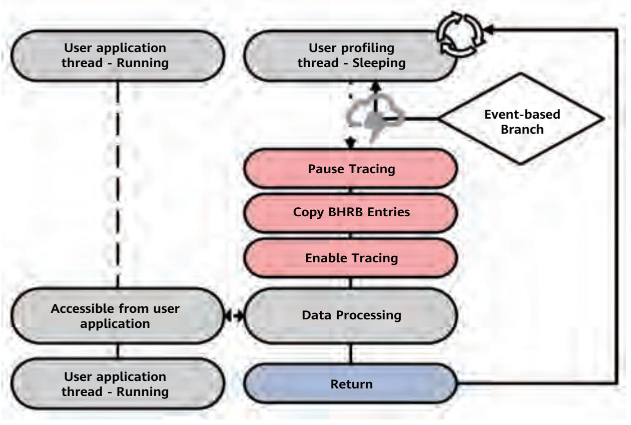

User application thread - Running
User profiling thread - Sleeping
Event-based Branch
Pause Tracing
Copy BHRB Entries
Enable Tracing
Accessible from user application
Data Processing
User application thread - Running
Return

Figure 7 Hardware tracing methodology to extract branch traces used in OpenJ9

and for tracing branch instructions in order to determine hot methods. First, the PMU is configured to randomly sample instructions. Then, the instruction addresses of the samples are mapped to methods and counted. The number of samples collected for a method provides a coarse indication of its hotness. Second, POWER8 provides an inexpensive mechanism for applications to obtain the addresses of the last branch instructions from the CPU core's Branch History Rolling Buffer (BHRB). OpenJ9 uses the addresses of jump instructions (`bl`) recorded in the BHRB to update the method invocation counts. The BHRB provides a buffer that stores the addresses of the last 32 branch instructions and their taken directions. The runtime retrieves the buffer entries through 32 special registers and a dedicated unprivileged machine instruction `mfbhrbe <register>, <BHRBEntry>`, without any need of context-switching.

Figure 7 shows the methodology to extract the branch history with minimal overhead. In this methodology, the application is continuously running, while a separate profiling thread is reading the BHRB and accounting each entry with the respective profile data. An event-based branch (EBB) wakes up the profiling thread when the BHRB is full. The profiling thread disables the BHRB to pause recording new entries while the thread copies the BHRB entries into the user-space buffer for further processing. The profiling thread then re-enables recording. Following that, the profiling thread processes the list of addresses for each branch instruction and increments the call count in the method data object of the called method. Then, the thread waits to be woken up by the next EBB. The advantage in the POWER8 architecture is that the interrupt handler (i.e., EBB) and the access to the buffer are in the user-space, and do not involve any overhead for privileged execution or context-switching. However, the BHRB is limited to 32 entries and cannot continue recording branches taken by the application thread while the BHRB entries are drained from the buffer by the profiling thread. This results in method invocations being counted inaccurately, which is one reason why OpenJ9 resorts to precise instrumentation-based counting in the highest profiling tier.

Similar techniques are provided by Intel and ARM processors and could be leveraged in an analogous way. However, because each technique has very constraining drawbacks, these techniques are not suitable for replacing the code instrumentation approach. Table 2 compares the different hardware mechanisms and their key characteristics.

Intel® Branch Trace Store (Intel BTS) [18] uses a similar approach as the IBM BHRB. However, the notification of the buffer state requires not only an OS-level interrupt but also elevated privileges in order to access the buffer. Therefore, the necessary OS-context switch may introduce an enormous overhead. Furthermore, Intel BTS has the same issue as IBM's BHRB — it may not capture all branches.

Intel® Processor Trace (Intel PT) is a tracing mechanism that captures information about software execution flow using a dedicated hardware facility with minimal performance overhead during tracing. The trace includes information on the direction of each control flow instruction and the elapsed time between them. The challenge in using Intel PT is the overly expensive processing required to decode the collected trace and to extract the relevant information. Such information may include executed branches in the JIT-compiled code, how often they were executed, and the taken directions. The JPortal project [19] integrated Intel PT in HotSpot JVM but uses it only for debugging and crash analysis purposes.

ARM CoreSight is heavily dependent on its implementation on the specific core. The main purpose is to provide debugging capabilities for embedded processors, while using dedicated off-chip hardware to capture the trace. Extending ARM CoreSight with new instrumentation and tracing capabilities may be a viable solution to offload the profiling to the hardware.

4.3 Proposed Hardware-Assisted Profiling

There are several possible solutions that can reduce the profiling overhead. A promising approach, similar to the one in OpenJ9, is using specialized instructions and buffers to record the last branches. Dedicating area in the CPU core for specialized hardware profiling units can reduce the profiling overhead or even enable zero-overhead solutions, which in turn open up opportunities for profiling even in the highest tier. While such solutions must not disturb or slow down the processing in the regular CPU core, they also need to offer an efficient software interface that can be used to update the profiling metadata in the runtime with minimal overhead.

The data profiles currently collected in HotSpot are different types of counts — such as invocation count, taken/not-taken counts in branches, and types used — as listed earlier in Table 1. The counts are updated in HotSpot JVM every time the method or the code for the corresponding bytecode instruction is executed. Primarily, an implementation that

Table 2 Existing hardware mechanisms for tracing of memory accesses and branches

	IBM POWER8	Intel BTS	Intel PT	ARM CoreSight
Accurate count	No	No	No	Yes
Interface	User-level accessible counters and interrupt	OS-level interrupt & privileged accesses	Hardware in chip exposed as PMU	External hardware
Timing information	No	No	Yes	Yes
Overheads	Minimal	Context switch Copying trace	Tracing: Low Decoding: High	Minimal Decoding: Offline
Filtering	Based on the type of instruction	Type of branch and jump instructions	Based on the address (2 address ranges)	Yes

uses hardware-assisted profiling must provide the same information to the JIT compiler as it currently obtains through instrumentation. The profiling hardware must support an arbitrary number of counters that count different types of events and that can be mapped to slots in the MDO. In some cases, however, it may be acceptable to use less expensive but approximate counters, such as the hardware implementation [20] of the SpaceSaving [21] algorithm for approximate frequent item counts.

One approach is to use the existing metadata MDO, in which the runtime stores the profile data. Figure 5e shows an example of an instruction that could be introduced and used for profiling in the JVM runtime. Such an instruction could replace the part of the code that is responsible for identifying whether the receiver type is already stored in a slot and could handle the incrementing of the respective counter. Such code is shown in red in Figure 5d. The necessary arguments for such an instruction are the number of distinct types to count and the type of the specific instance (type of `obj[i]` is identified by the register `rdi`). The location of the metadata MDO (address `0x7f9be3caa918`) is known by the JIT compiler when native code is generated and passed in another immediate field of the instruction.

When the CPU core executes this new instruction, the requested profiling information is written into the MDO by a specialized profiling unit that can be loosely coupled with the core. That is, after sending the tuple (increment, `rdi` value, slot address) to the profiling unit that is coupled to the core through a FIFO queue, the core can resume executing application instructions while the profile units execute the increment operation similar to a remote-atomic instruction. By using such an approach, the profiling is outside of the critical path of the execution. Furthermore, replacing the 23 profiling instructions shown in red in Figure 5d with one instruction also lowers the front-end pressure on the core.

We can expect such an instruction to radically reduce the overhead, meaning profiling could be enabled in the highest compilation tier. This continuous collecting of profiling information would allow the runtime to detect changes in the application behavior, such as changes in branch frequencies or encountered dynamic types. When the runtime detects that currently collected profile has significantly deviated from the profile that was used to generate the currently running code, it can request a recompilation of the code. The JIT compiler can then leverage the most recently collected profile data to produce code that is optimized for the current application behavior. Continuous profiling in JIT-based runtime allows continuous optimization of the executed code. Furthermore, profiling could be completely disabled by replacing the profiling instruction with a NOP instruction or by disabling it through setting a hardware mask.

5 Speculation: Cost When JIT Is Right

JIT compilers often make assumptions on a program's control-flow, types, and values in order to generate code that is more efficient. For example, they might omit special cases assumed not to be encountered during execution. However, in return for being able to create more efficient code, the compiler must include checks in the code to verify that the assumptions hold at run-time. In the unlikely case they fail, the program execution must halt and trigger a deoptimization event. With the new insight gained from the failed execution, the compiler then recompiles the loop or function with the failed assumption removed. For dynamically typed languages, speculation on types is crucial for obtaining a significant performance gain over threaded interpretation [22]. In this section, we discuss type speculation for JavaScript in the V8 engine and describe an ISA extension [23] (representative for other types of checks) in which the frequent *SmallInteger (SMI)* checks are offloaded to hardware.

5.1 Overhead of Checks in V8

The additional instructions that implement checks introduce a run-time overhead, regardless of whether the checks fail or succeed. In our previous work [23], we analyzed this overhead for the V8 JavaScript engine on the JetStream2 benchmark suite. We found that about 4% of all the machine instructions emitted by the compiler were used to implement checks. Put differently, there is a check every 20–25 instructions. In order to quantify their overhead in the noisy environment of the V8 engine, we followed a statistical approach and combined two different methods. In the first method, we used time-based sampling and determined the overhead as the fraction of samples that fall on instructions that are part of a check. In the second method, we measured the overhead directly by modifying the compiler to prevent the emission of checks in places where we knew they would never trigger a deoptimization in the studied workloads (removing all checks is impossible because doing so would break the programs). Both methods are approximate. In the first case, an instruction may be used as part of a check as well as in regular application code. The second case is approximate since not all checks could be removed. By combining the results obtained from both methods and applying statistical significance testing, we found an average run-time overhead of 8% due to the checks, reaching up to 20% in extreme cases.

5.2 Type Speculation in JavaScript

Consider the following JavaScript function[4]:

```
function add(a, b) {
  return a + b
}
```

The function can be called with values of different types, for example, numbers and strings. The ECMAScript [24] specification requires that numbers are treated as values in double-precision floating-point. However, V8 — like other JavaScript engines — uses an optimization for SMI values, treating them as ordinary integers. This means that, if a and b are SMIs, the operation is an integer addition. And if one of the arguments is a floating-point number, the operation is a double-precision addition. However, if the arguments are strings, the + operation is a string concatenation.

Assume that the function has been called a number of times already while the code is executed in the V8 interpreter and, as such, it has become sufficiently hot to be compiled by the V8 JIT compiler. During execution in the interpreter, V8 keeps track of the actual type of the + operation. Also assume that the operation was always performed as an SMI addition (i.e., both operands were small integers). Based on this information, the compiler speculates that the function will continue to be called this way, and assumes that the parameters a and b will remain SMI values.

For the following description, we use V8 and enable the memory optimization that uses 32-bit compressed pointers on 64-bit platforms. With this optimization, pointers and SMIs are represented as tagged values in a 32-bit word. The least significant bit in a tagged value is used to discriminate between pointers and SMIs. In this case, if the least-significant bit is zero, the remaining 31 bits are interpreted as a signed 31-bit integer value; otherwise, the tagged value is a compressed pointer.

When generating the code for function `add(a,b)` and speculating that both parameters `a` and `b` are SMIs, the compiler needs to emit checks to verify that the values in `a` and `b` have their least-significant bits cleared. If one or both of the parameters have the bit set, the compiler needs to emit code that triggers a deoptimization event. This is because the speculation has failed.

Figure 8a shows the ARM64 code for the function generated by V8's JIT compiler. The instructions shown in red are the SMI checks for the two arguments passed to the function on the stack. The SMI check tests the least-significant bit and branches to the corresponding deoptimization trampoline. Here, we name arguments `a` and `b` as `a_not_smi` and `b_not_smi`, respectively, for illustrative purposes. If the least-significant bits of both parameters are zero, an untagging arithmetic right shift is applied to convert SMI into a 32-bit machine integer.

5.3 Hardware Offload of Guard Checks

V8's JIT compiler uses 52 different types of checks. The checks that have the largest execution overhead are SMI and bounds checks in array accesses in the JetStream2 benchmark. Together, these checks account for approximately 50% of the overhead. Further analysis shows that the overhead does not originate from the branch instruction of the check. That is not surprising because the branches are only taken if the checks fail. Given the conservative speculations by the compiler, such failure happens only rarely. This means that the branches can be accurately predicted.

However, we believe that the use of branches here is not appropriate because, semantically, the checks more closely resemble assertions. Once triggered, execution will take an exception-like path into the runtime that handles the deoptimization. However, we found that removing the branches while leaving the instructions that compute the branch condition untouched provides only a small speedup of 1–2%. We also found that this speedup is solely due to the improvement of the misses on the application's

[4] Because this example is for illustration only, we disabled inlining and on-stack replacement optimizations to force the compiler to generate a function for this addition operation.

```
; in add() function
...
ldr   x2, [sp, #56]       ; x2 <- parameter a
tst   w2, #0x1            ; test lsb of w2
b.ne  a_not_smi           ; not cleared -> deopt: a not a SMI
ldr   x3, [sp, #64]       ; x3 <- parameter b
tst   w3, #0x1            ; test lsb of w3
b.ne  b_not_smi           ; not cleared -> deopt: b not a SMI
asr   w4, w2, #1          ; w4 <- w2 >> 1
adds  w4, w4, w3, asr #1  ; w4 <- w4 + (w3 >> 1)
...
```

(a) Original ARM64 code with SMI checks

```
; runtime call to set bailout handler address
adrp      x0, bailout_handler
add       x0, x0, :lo12:.bailout_handler
msr       REG_BA, x0

; in add() function
...
jsldursmi w2, [sp, #56]    ; w2 <- load_smi(a)
jsldursmi w3, [sp, #64]    ; w3 <- load_smi(b)
adds      w3, w2, w3       ; w3 <- w2 + w3
...

; in bailout_handler
.bailout_handler:
mrs       x0, REG_PC       ; x0 <- PC failed load
mrs       x1, REG_RE       ; x1 <- bailout reason
b         runtime_bailout  ; call into runtime
...
```

(b) ARM64 code with the `jsldursmi` offload instruction

Figure 8 (a) ARM64 assembly code generated by V8's JIT for the example function speculating that both parameters are SMI values. Instructions for SMI checks are shown in red. (b) ARM64 assembly code generated by V8's JIT using the ISA offload instruction. Instructions and registers highlighted in green need to be introduced to support the proposed ISA extension.

branches because this removes 20% of the overall retired branches. In order to further reduce the overhead, we must also try to offload the computation of the check condition, the assertion, and the format conversion.

We proposed a small ISA extension and demonstrated it for SMIs in [23]. It is based on two ideas. The first idea is that SMIs are a memory optimization. Values are represented in SMI form when they are stored as fields of objects in the heap. However, as soon as they are loaded into a register, they should be converted into machine integers and remain in this format. Only when a machine integer needs to be written back into a heap object should it be converted back into the SMI representation. The second idea is to treat the check as a trap, which could then be handled in user-space. Our ISA extension consists of the specialized SMI-load-check-and-convert instructions `jsldrsmi` and `jsldursmi`, corresponding to the ARM64 `ldr` and `ldur` regular load instructions. They load the value, check that its least-significant bit is zero, and perform the untagging right shift. If the check fails, the core's exception mechanism is used to jump to a user-space bailout handler at instruction commit. Three additional machine-specific registers are introduced: REG_PC, which upon entry of the bailout handler contains the PC of the failed SMI-load and REG_RE, which contains the code of the bailout reason (similar to a syndrome register). The runtime sets the REG_BA register to the address of the bailout handler, which can handle deoptimization for several compiled functions. It reads the REG_PC and REG_RE, and calls into the runtime what is currently being done in the trampoline code (at `a_not_smi` and `b_not_smi` in the example above).

Figure 8b shows the compiled `add(a,b)` function with the initialization code that sets up the address of the bailout handler, and the code for the bailout handler itself. The ISA extension has a relatively low implementation complexity. Conceptually, it consists of a low-overhead data-path extension with a constant arithmetic shift, two multiplexers, and three additional registers.

We implemented the extensions in the gem5 simulator and measured an average speedup of about 3% from the SMI ISA extension and a speedup of up to 10% in SMI-heavy micro-benchmarks. Although the ISA extension is specific to V8, the instruction could be adapted to support value tagging used by other JavaScript engines. For example, it could support NaN-Tagging, which encodes bits in the redundant encoding of FP64 NaN values. A similar approach could also be applied to the other 51 checks in V8 (i.e., bounds, overflow, and type checks). From our analysis on the JetStream2 benchmark suite, we conclude that there is a potential speedup of 8% in the ideal case where *all* checks are offloaded and their implementation has zero run-time overhead.

6 Speculation: Cost When JIT Is Wrong

The mechanism responsible for guaranteeing correctness and continuity when a compiler speculation proves to be wrong is present under different names in the JIT compilers we discussed earlier. In HotSpot JVM, the term used for this situation is *deoptimization,* or *deopt* for short. Deoptimizations are expensive operations that require translating the machine thread state (i.e., the call stack with the machine registers) into the abstract program state of the VM. More precisely, if an assumption made by the compiler no longer holds, the execution of the generated code needs to stop, the generated code must be discarded, and the execution must be resumed at a lower tier. In the case of the HotSpot JVM, the execution resumes in the Interpreter and, oftentimes, the method needs to be re-profiled in order to tier up again.

The cost involved in deoptimization comes from multiple sources. These sources include the following: saving the current state, running interpreted code instead of highly optimized machine code, and the lag to tier up and reoptimize the code in question.

Consequently, most environments with JIT compilers are rather reluctant to employ speculation, even though it may potentially generate code that is more efficient. In particular, the JIT compiler in HotSpot JVM mostly speculates only when it is fairly certain that the assumption will hold. As such, many opportunities for optimizations that would require more aggressive speculation are missed. The extra performance they would have provided is the price paid for the deoptimization mechanism.

6.1 The Anatomy of Deoptimization

Deoptimizations can be *synchronous* or *asynchronous*. The former refers to those caused by an invalid assumption in the current thread, for example, a branch that is taken only once when the program execution is already stable. In HotSpot parlance, synchronous deoptimizations are also called *uncommon traps*. Asynchronous deoptimizations are usually requested by another thread, for example, when a subclass is loaded and the assumptions that allowed a virtual call to be devirtualized no longer hold, resulting in the need for deoptimization.

In the HotSpot JVM, a compiled method could be in any one of the states summarized in Table 3. Briefly, all compiled code starts out as *not-installed*, being just an entry in the *code cache*. The code cache is a memory area where generated native code is located. When code is in the not-installed state, it is not yet associated with any particular method. When the code is registered with a method (i.e., a thread calls said method and executes this code), the state of the compiled code becomes *in-use*. Ideally, this code is optimal and will execute for the rest of the program. However, it normally takes a few optimization–deoptimization iterations before the code is in its final state.

Table 3 The lifecycle of a compiled method in HotSpot JVM. A method is *alive* when the code is ready to be used by the application threads.

State	Description
Not-installed	The compiled code exists in the code cache, but it is not yet registered with the method.
In-use	The compiled method is in use and the code is correct; new threads can use this version of compiled code.
Not-entrant	This version of the code is not valid anymore. The method is still alive.
Zombie	The invalid code is not used by any thread anymore.
Unloaded	The code is unregistered and cleaned up from the code cache.

When a method needs to be deoptimized, it is marked as *not-entrant*. The thread that triggered the deoptimization continues its execution in the interpreter until it is ready to run optimized code again. No new threads are allowed to call to the not-entrant code. Threads that were already in the process of executing the now-not-entrant code are allowed to finish executing it. Before resuming the execution in the interpreter, the thread that transitioned the method into the not-entrant state also notifies a special *sweeper thread* of this state change. The sweeper thread is responsible for garbage-collecting the code cache.

The code cache is swept asynchronously with respect to the application threads that are switching between optimized and interpreted code. The sweeper thread is activated when the free space in the code cache drops below a certain threshold and checks each compiled method in turn — if the code is still in use by any thread, the sweeper thread marks the method as *active*. If the method is reported as not-entrant and the sweeper thread finds it active, the sweeper does not change the method's state. Otherwise, it transitions the method to *zombie* state. A zombie method is off-limits to all application threads. Finally, with the next iteration, the sweeper *unloads* the compiled method and removes the native code from the code cache.

6.2 Estimating Deoptimization Overheads

Quantifying the true costs of deoptimizations is challenging due to the complexity that arises from the number of steps and threads involved. Besides the application thread's effort to translate the stack frames from native to interpreted, there is also the overhead of running in the interpreter until the code is recompiled. We could also try to account for the overhead of the sweeper thread: although it wakes up only occasionally and does not block the application, the sweeper does share resources with the application and intuitively should have an impact. Moreover, not all deoptimizations precisely follow the same typical sequence of events: depending on various internal heuristics, the method may or may not need to be reprofiled, recompiled (i.e., old code in the code cache is reused), etc. As such, different optimization reasons trigger deoptimization events with different costs. In addition to all of this, the workload itself influences the perceived cost of a deoptimization. The same deoptimization process will be seen as being more expensive for a simple application than for a long-running massive one.

A first step toward quantifying the cost of deoptimizations is to use a controlled environment in which deoptimization events can be explicitly triggered and their overhead measured. We employ a simple synthetic micro-benchmark that calls a method one million times in a loop. The method contains a conditional statement for which one of the branches is taken only once during the execution of the benchmark. This results in the native code being optimized to always take one branch. When the other branch is taken, an uncommon trap is triggered and, subsequently, the deoptimization process, that we want to measure, is also triggered. In order for the deoptimization event to be easily observable, we force the compiler to not inline the method. By nesting conditional statements and triggering uncommon traps every 2×10^5 iterations, we are able to control the number of deoptimizations. We specifically choose the interval between deoptimization events so that the code barely has time to tier up between any two deoptimizations. As soon as the code is compiled at Tier 4 again, a new uncommon trap is triggered. This is an adversarial scenario intended to stress the deoptimization process.

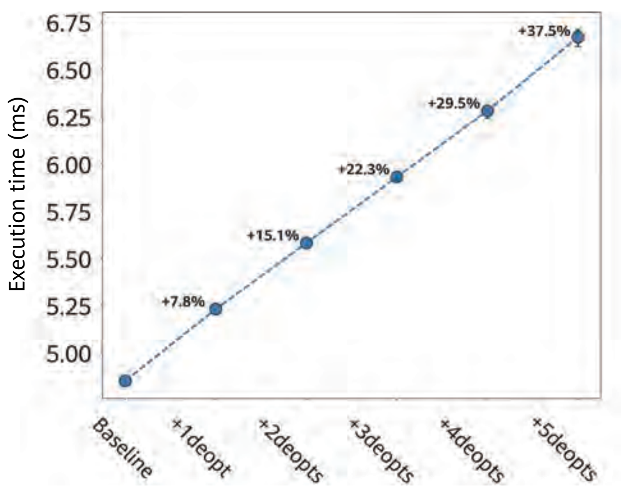


Figure 9 Overhead of deoptimizations as measured on a synthetic micro-benchmark. The Y axis represents the execution time of the main loop for the baseline (without deoptimization) and the time with up to five additional deoptimizations, which are marked on the X axis. The relative overhead with respect to the baseline is indicated above each data point.

As a baseline for the benchmark, we consider an execution that never deoptimizes the method in the main loop (i.e., the conditional statement inside the method always takes the same branch). We define the deoptimization cost as follows: the execution time overhead for the main loop of the benchmark when deopts occur compared with the baseline. We measure the overhead of up to five additional deoptimizations and execute each experiment 11 times. The results are stable across runs (see Figure 9). A single extra deoptimization incurs an overhead of about 7%, increasing almost linearly up to 37% for five deoptimization events.

It is important to note that the observed overhead is a function of the benchmark's complexity. In particular, the workload in our micro-benchmark can be made arbitrarily large, in which case the overhead of the deoptimization process will be perceived differently. Most of the time, real-world applications do not have such an adversarial approach to deoptimizations as the one we employ in our synthetic design. Although the number of deoptimization events is normally much higher, these events are not triggered continuously back-to-back on a piece of code. Figure 10 gives an idea of the number of deoptimizations in more realistic scenarios, illustrating this on the Renaissance benchmark suite. The figure represents the distribution of deoptimization events over 11 runs with default parameters for the runtime and benchmarks. We observe that almost half of the applications have over 1,000 deoptimization events.

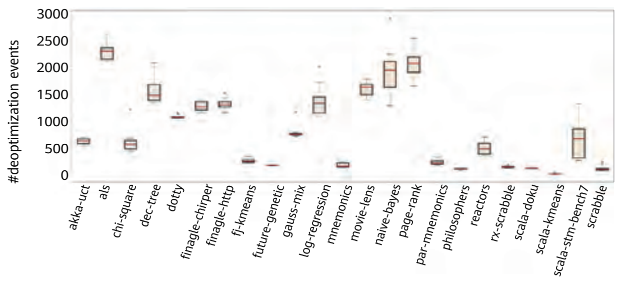


Figure 10 Distribution of the number of deoptimization events in the Renaissance benchmarks (data collected over 11 runs)

In a real-world application, part of the cost will potentially be hidden by the overall complexity of the execution. However, a next-to-linear increase in overhead with the number of deoptimizations, in addition to the activity of the sweeper thread, could eventually develop into a notable performance penalty. If the number of deoptimization events increases significantly, the cost may further become prohibitive. Due to this concern, the JIT compiler is prevented from engaging in speculation that is more aggressive and susceptible to generating large numbers of additional deoptimizations.

6.3 Opportunities for Hardware-Software Co-Design

When applying speculative techniques, the gains obtained through successful speculation must be balanced against the costs or penalties that occur when it fails. Here, we define gains as performance improvements, and costs as the additional run-time overhead resulting from deoptimizations. Speculation is beneficial if the following inequality holds:

$$\mathrm{P(right)} \times \mathrm{PerformanceGain} \geq (1 - \mathrm{P(right)}) \times \mathrm{DeoptCost} \quad (1)$$

Assuming that the speculation of interest has a success probability of $\mathrm{P(right)}$ and achieves a gain of $\mathrm{PerformanceGain}$, the only parameter left to control is the cost when the speculation fails, $\mathrm{DeoptCost}$. If we can lower this cost, we can tolerate speculations that are more aggressive. We can then consider optimizations that have a lower success probability $\mathrm{P(right)}$. One contributor to the cost of deoptimizations is the translation of the machine execution state into the abstract virtual machine state used by the interpreter (e.g., the translation of the machine stack frames to stack frames of the JVM). A hardware-software co-design approach to accelerate the switch from native to interpreted execution could be achieved by storing the machine state (e.g., stack frames and the content of CPU registers) in a format that can be directly used in the interpreter — the runtime would need to be modified to handle this new format. The challenge here is developing a representation format that is both simple to create in hardware and efficient for the interpreter to use.

The proposed hardware-software approach should also be used in the reverse direction. For example, it should be used for an OSR, in which a loop is compiled to native code before the entire method is compiled. In this case, the interpreter state needs to be converted into the native machine state.

7 Speculation from Value Profiles

In Section 5, we described how the V8 JavaScript engine can produce efficient code despite the dynamically typed system of JavaScript. The V8 engine collects type information during execution and its JIT compiler uses this information to generate specific code for the encountered types. This code is considerably faster than the generic code in which the individual, typically untyped, bytecode instructions are implemented. Such optimization can be regarded as a specialization that is based on types. Other specializations beyond types are also possible. In this section, we describe the specialization based on values.

Listing 2 Function deployed on a serverless platform. The `recommend` method can be specialized by the JIT compiler from the value profiles of the method arguments. For example, if two arguments are found to be constant, the specialized method takes only two arguments instead of four.

```
public class RecommendFunction {
  private Backend backend;

  public Recommendation[] handleRequest(Request
  req) {
    return Recommendation.from(
      backend.recommend(  // invoke backend
      req.userId, req.requestId,
      req.numSuggestions, req.fromMobile));
   || found always    =10    and    =true
  }

  static record Request(
    String userId, int requestId,
    int numSuggestions, boolean fromMobile) { }
    ...
}
```

Take the Java code in Listing 2 as an example. It contains a snippet of the request handling function of a recommender microservice that could be deployed on a Function-as-a-Service (FaaS) platform. The functions are exposed through a Web API, and the arguments for the requests are typically submitted as JSON payloads in HTTP requests. In this example, we assume that the FaaS hosting platform is parsing the JSON data into a Java container object of type `Request`, and it then passes this object to the `handleRequest` method that implements the service. The use of such request objects has the advantage that they can represent a large number of arguments, some of which can also be optional. In the example, the values of fields in the request are passed to the backend `recommend` method in a regular argument list.

Let us further assume that most of the requests originate from a mobile application that always fetches 10 recommendations, such that numSuggestions=10 and fromMobile=true. If the JIT compiler knew this, it could create a specialized version of the `recommend` method that takes only two arguments but with the parameters numSuggestions=10 and fromMobile=true fixed to constants. This specialization is also known as partial evaluation [11, 25] in which a program is specialized to part of the input.

Today, this specialization requires guidance from developers through annotations in the source code [11, 26]. Lima et al. [26] argue that developer guidance is needed because of the overhead of profiling values in the program and discovery of good profiling opportunities. However, we believe that the problem of expensive value profiles can be addressed by a hardware profiling unit that counts the distinct values and critical places in the code (such as at inlining boundaries), for example, in `handleRequest`.

The availability of a value profile offers new optimization opportunities for the JIT compiler. Lima et al. [26] reported a 2.5× speedup from properly placed specialization annotations in their implementation in the JavaScriptCore engine.

8 Conclusion

Up to this point, we have discussed the different aspects that impact the performance of managed code in a runtime system, and we have outlined possible hardware-based enhancements separately for each of them. Figure 11 shows how all the pieces fall into place to form a balanced hardware-software co-design platform for dynamically compiled languages.

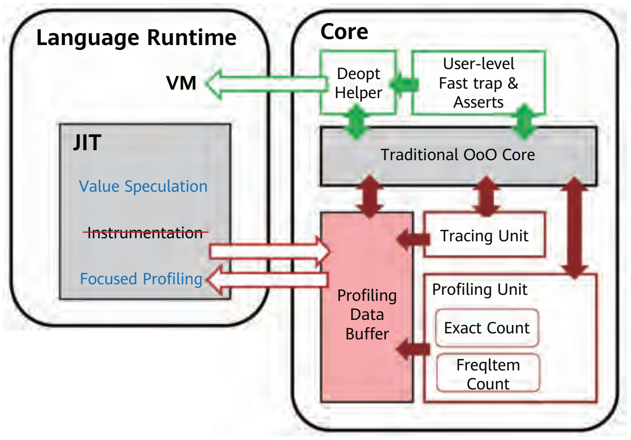


Figure 11 Proposed architecture for a hardware-software co-designed execution platform for dynamically compiled languages. Hardware extensions dedicated for profiling and speculation are added to typical CPU cores. The software directs the actions of the hardware and consumes the collected information as needed.

On the hardware side, we build on top of an existing state-of-the-art CPU core. To this, our design adds a number of hardware extensions, with the goal of improving the execution of JIT-compiled code and the functionality of the JIT compiler itself. The right-hand side of Figure 11 illustrates the hardware design. The components in the bottom part of the figure (shown in red) represent the newly proposed profiling infrastructure. The extensions are intended to incur minimal overhead and to permit narrowly focused profiling that can be steered from software. On the one hand, we envision profiling units that enable fine-grained control over specific functions or loops, based on ranges of instruction addresses. To this end, current PMU functionality, as presented in Section 4, could be adapted to provide the necessary hardware information for our targeted profiling goals. On the other hand, we are considering new units responsible for aiding with value profiling and tracing. The latter can be used for trace-based JIT compilers.

The components shown in green, on the top side of the hardware design area, represent extensions that are involved in lowering the cost of speculation. We consider both overhead-inducing scenarios: when speculation is correct (as described in Section 5), and when the assumptions turn out to be wrong (addressed in Section 6). For the former, the costs are caused by always executing guard checks that succeed most of the time and do not contribute to the progress of the application code. We count on a hardware unit that enables the use of user-level trap handlers and assertions to reduce the amount of overhead incurred from this source. The latter specifically targets deoptimization costs. These costs could be reduced with a hardware-assisted solution that captures the machine state (e.g., stack frames and registers) and converts it into an abstract state representation of the virtual machine.

The integration with a language runtime is represented by the left-hand side box in Figure 11. We modelled our approach on the functionality of HotSpot JVM. First, the existing instrumentation that provides profiling information should be replaced with the hardware-based profiling extensions. The expected outcome of this measure is twofold: (1) profiling at lower-level tiers, such as Tier 3 in HotSpot, will incur virtually no cost, and (2) profiling at the highest-level tier, such as Tier 4 in HotSpot, will become possible. The latter outcome also facilitates other types of optimization, such as addressing the issue of outdated profiles. In our approach, the VM must control the hardware profiling units and instruct them on what information is currently relevant for profiling and what information should be collected. For example, the hotness of functions and loops should be profiled first. The focus could then turn to collecting back-edge counts and finally to addressing value profiles. Subsequently, value profiles can be used in a new component that implements value speculation as part of the compilation process. This will enable frequently executed functions and loops to be specialized for particular values.

Although we describe a solution primarily based on the JIT compiler model of HotSpot JVM, our vision extends toward a unified approach that can be leveraged by other languages and runtimes as well. Overall, we propose a hardware-software design for JIT-based runtimes, where the hardware takes advantage of informed software just as much as the software benefits of informed hardware. We believe that such a design would be able to provide the perfect balance and interplay between typically disparate or opposed aspects of a runtime system: compilation and execution, performance and overhead, and software and hardware.

References

[1] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: PyPy's tracing JIT compiler," in *Proc. of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS'09)*, 2009, pp. 18–25.

[2] C. Wimmer and S. Brunthaler, "ZipPy on Truffle: A fast and simple implementation of Python," in *Proc. of the 2013 Companion Publication for Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'13)*, 2013, pp. 17–18.

[3] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proc. of the 2003 IEEE/ACM Int'l Symposium on Code Generation and Optimization (CGO'03)*, 2003, pp. 265–275.

[4] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05),* 2005, pp. 190–200.

[5] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the JVM," in *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, 2019, pp. 31–47.

[6] E. Duesterwald, "Design and engineering of a dynamic binary optimizer," in *Proc. of the IEEE*, vol. 93, no. 2, pp. 436–448, 2005.

[7] R. Zhou and T. M. Jones, "Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation," in *Proc. of the 2019 IEEE/ACM Int'l Symposium on Code Generation and Optimization (CGO'19),* 2019, pp. 15–25.

[8] D. Chen, D. X. Li, and T. Moseley, "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," in *Proc. of the 2016 Int'l Symposium on Code Generation and Optimization (CGO'16)*, 2016, pp. 12–23.

[9] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A practical binary optimizer for data centers and beyond," in *Proc. of the 2019 IEEE/ACM Int'l Symposium on Code Generation and Optimization (CGO'19)*, 2019, pp. 2–14.

[10] J. Laskey, "Adventures in JSR-292 or how to be a duck without really trying," in *Java Language Summit (JVMLS'11)*, 2011.

[11] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *Proc. of the 2013 ACM Int'l Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*, 2013, pp. 187–204.

[12] M. Rigger, M. Grimmer, and H. Mössenböck, "Sulong – execution of LLVM-based languages on the JVM: Position paper," in *Proc. of the Workshop on Implementation, Compilation, Optimization of ObjectOriented Languages, Programs and Systems (ICOOOLPS '16)*, 2016, pp. 1–4.

[13] R. Mueller and L. Stadler, "grCUDA: A polyglot language binding," in *CodeOne*, 2019.

[14] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proc. of the ACM Symposium on Cloud Computing (SoCC'18)*, 2018, pp. 199–211.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, 2006, pp. 169–190.

[16] A. Mericas, N. Peleg, L. Pesantez, S. B. Purushotham, P. Oehler, C. A. Anderson, B. A. King-Smith, M. Anand, J. A. Arnold, B. Rogers, L. Maurice, and K. Vu, "IBM POWER8 performance features and evaluation," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 6:1–6:10, 2015.

[17] M. Gschwind, "Perking up code: Architecting high-performing dynamic compilation systems for the future," in *Proc. of the 3rd Inter'l Workshop on Dynamic Compilation Everywhere (DCE-3)*, 2014.

[18] I. Corporation, "Intel® 64 and ia-32 architectures software developer manuals," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/ articles/technical/intel-sdm.html

[19] Z. Zuo, K. Ji, Y. Wang, W. Tao, L. Wang, X. Li, and G. H. Xu, "JPortal: Precise and efficient control-flow tracing for JVM programs with Intel processor trace," in *Proc. of the 42nd ACM SIGPLAN Int'l Conference on Programming Language Design and Implementation (PLDI'21)*, 2021, pp. 1080–1094.

[20] J. Teubner, R. Mueller, and G. Alonso, "FPGA acceleration for the frequent item problem," in *Proc. of the 26th IEEE Int'l Conference on Data Engineering (ICDE'10)*, 2010, pp. 669–680.

[21] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. of the 10th Int'l Conference on Database Theory (ICDT'05)*. SpringerVerlag, 2005, pp. 398–412.

[22] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, "The structure and performance of interpreters," in *Proc. of the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, 1996, pp. 150–159.

[23] A. Parravicini and R. Mueller, "The cost of speculation: Revisiting overheads in the V8 JavaScript Engine," in *Proc. of the 2021 IEEE Int'l Symposium on Workload Characterization (IISWC'21)*, 2021, pp. 13–23.

[24] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed., June 2011.

[25] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Code Generation*, 1st ed. Prencie Hall, 1993.

[26] C. Lima, J. Cezar, G. Vieira Leobas, E. Rohou, and F. M. Quintã o Pereira, "Guided just-in-time specialization," *Science of Computer Programming*, vol. 185, p. 102318, 2020.

Data-Centric Auto-Tuning of High-Performance Computing Applications

Baicheng Yan [1], Giulio Stramondo [2], Zongyan Cao [1], Zhe Wang [2], Long Wang [1, *]
[1] Computing System Optimization Lab
[2] VonNeumann Lab Zurich

Abstract

Performance portability has always been an important aspect in the field of high-performance computing (HPC). Today, due to the rapid evolution of computation architectures, maintaining good performance on different target systems has become essential. The use of data-centric representations has shown the potential for improving performance while maintaining the portability of an application. In this paper, we present DCTuner, an auto-tuning method for optimizing HPC applications using a data-centric representation composed of our pruning and exploration algorithm. Our experimental results show that DCTuner improves the performance of real-world HPC applications compared to the state of the art by up to 14.67%, while maintaining a high degree of portability.

Keywords

auto-tuning, high-performance computing, dataflow, performance optimization, portability

* Corresponding author

1 Introduction

The pursuit of better performance often results in less portability [1, 2]. Rapid changes in architectures and computing systems are forcing users to modify the code in order to take advantage of new architectural features [3–5]. Such changes often specialize the code for a specific architecture and corrupt the application code, making it harder for developers to maintain it. Thus, *performance portability*, the ability of an application to deliver good performance on various architectures without any code modification, is becoming increasingly important in the high-performance computing field, as it could provide benefits to both application's maintainability and performance, while reducing development costs.

Two main components are generally required to realize a performance portable framework: an abstract representation of the application that captures all the relevant information — e.g., data movement, data dependencies — and a method to optimize such a representation given a target architecture.

Recent performance portable approaches, given the significant impact that data movements have on energy and performance in modern systems [6], adopt code representations that explicitly describe the data movement of an application. These **data-centric** representations can facilitate analysis and optimization of applications: leveraging the dataflow model, in which dependencies among operations are represented explicitly, parallelism opportunities emerge naturally, giving additional freedom to optimization tools. Some of these approaches evolved in the context of Machine Learning (ML) frameworks, such as TensorFlow [7], MetaFlow [8], TVM [9], TASO [10], while others were designed for the purpose of optimizing HPC applications [2, 11–13].

The optimization of a performance portable application for a set of target architectures can be performed manually or automatically. Manual optimization has the potential to achieve the best possible performance across various architectures; however, it requires expert knowledge of the platform and is time consuming.

Among the automatic optimization strategies **auto-tuning** has been shown to be a viable option to automatically tailor an application to a target architecture [14]. There are various types of auto-tuning methodologies. In the simplest methodologies, a user provides a set of parameters and a range of values for each parameter, the auto-tuner then explores the parameter space looking for the near-optimal combination of parameters [14]. Some optimization frameworks like DaCe allow an external tool to perform auto-tuning on a set of code transformations. The realization of an auto-tuning method that explores a space of code transformations needs to overcome two main challenges: the exploration of an *implicitly defined* solution space, and the detection and avoidance of equivalent sequences of transformations. Due to possible dependencies among transformations, the solution space to be explored by the auto-tuner might not be completely determined statically, because some transformations might be enabled after the application of an enabler transformation. Different sequences of transformations might generate equivalent implementations — *redundant* sequence of transformations. Or worse, a sequence of transformations might be **cyclical** — meaning that, starting from the implementation i_1 of an application, after applying a sequence of transformations $t_1, \ldots, t_n$, we obtain an implementation i_n equivalent to i_1, on which we could re-apply indefinitely the transformations $t_1, \ldots, t_n$.

To achieve better performance and maintain portability, in this work we present **DCTuner**, a method to auto-tune HPC applications integrated with the DaCe framework [2]. Our method is able to explore an implicitly defined transformation space performing **deep optimization**: progressively applying transformations to the initial implementation of an application, and recomputing after each transformation the new set of available transformations. Deep optimization allows DCTuner to explore a wider space of transformations compared to traditional methods that define statically the set of available transformations [8, 10], **achieving a performance gain of up to 86%, when compared with using DaCe's automatic transformations**. Moreover, to address the problem of redundancy and cyclic transformations, we propose a **pruning strategy** which reduces the set of transformations evaluated by removing redundant transformations and allows deep optimization to be used in practice.

In summary, the main contributions of this work are the following:

- An auto-tuning method (Chapter 3), able to evaluate combinations of iterative transformations — deep optimization, based on our greedy search algorithm (Section 3.1).
- A pruning strategy that effectively reduces the search space and guarantees the convergence of our approach (Section 3.2).

- Extensive validation of DCTuner performed through three experiments: (1) comparison of our heuristic algorithm against extensive and random exploration of the transformation space (Section 4.2); (2) demonstration of performance portability using two different target architectures (Section 4.3); (3) improvement of DCTuner over the DaCe baseline on a real-world HPC kernel (Section 4.4).

2 Deep Optimization and Its Challenges

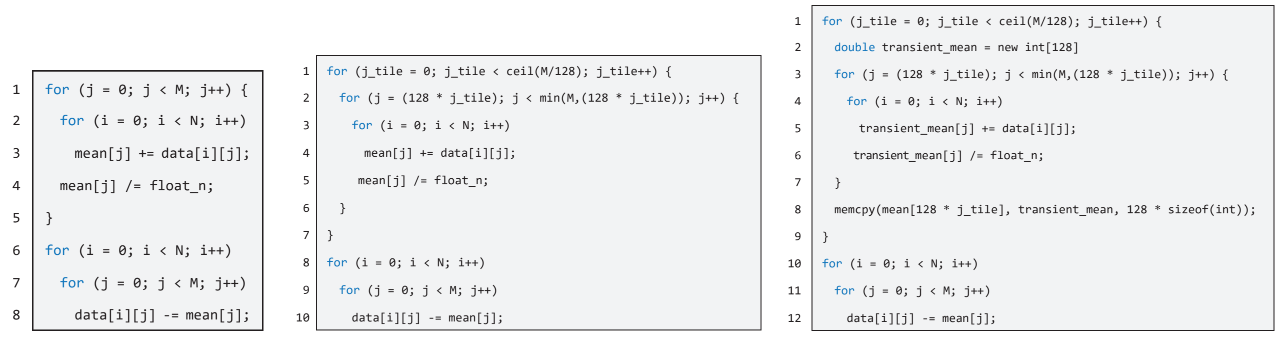
We refer to the example in Figure 1 to describe the problem we address using deep optimization. Figure 1a shows the original implementation of the *covariance* kernel from PolyBench [15]. After analyzing the original code looking for patterns that can be optimized — e.g., using the DaCe Framework — we select, among other possible candidates, two transformations that we take as example: transformation 1, applying tiling on the loop nest at *line 1*, and transformation 2, applying tiling to the loop nest at *line 6*. If we apply transformation 2 to the original code, the resulting implementation will suffer a 50% slowdown compared with the original code. Instead, if we apply transformation 1, we obtain the code shown in Figure 1b. We can then perform a new analysis on the code in Figure 1b and the result would show that a new set of transformations was enabled after the application of transformation 1. One of these new transformations, introducing a tile-sized transient for the *mean* array, shown in Figure 1c, once applied, leads to a 130% improvement over the original implementation.

Deep Optimization is a methodology to explore the transformation space in which the set of possible transformations is re-computed after each transformation is applied. Using this methodology, the sequence of transformations shown in Figure 1 can be discovered. Related auto-tuning tools, without using deep optimization, would only explore the set of transformations directly applicable to the original implementation in Figure 1a, and would not be able to find the combination of transformations leading to the implementation in Figure 1c.

2.1 Challenges

The example in Figure 1, discussed in Chapter 2, showed that it is not possible to generate the *deep* transformation space — the space composed by the **iterative combination** of multiple transformations — by analyzing only the initial kernel implementation. This happens because a transformation might require a previous *enabler* transformation to be applicable. It is, therefore, required to analyze the transformed code and generate a new set of enabled transformations after each transformation is applied to explore the deep transformation space, making this an *implicitly* defined solution space [16].

There are two main challenges that need to be addressed in order to efficiently explore the *deep* transformation space. The first challenge comes from the fact that the exploration of an implicit transformation space requires analysis steps to be performed, from every point of this space, to generate the next set of possible transformations. The increased amount of computation required, combined with the increased size of the explored space makes the exploration of the deep transformation space expensive. To address this problem, we propose a greedy algorithm (Section 3.1) that focuses the exploration of the transformation space by analyzing only the paths that are more likely to lead to the best solution. The second challenge arises from the possible existence of cycles in the transformation space that might cause an exploration strategy to loop endlessly when applying inverting transformations.



```
1  for (j = 0; j < M; j++) {
2    for (i = 0; i < N; i++)
3      mean[j] += data[i][j];
4    mean[j] /= float_n;
5  }
6  for (i = 0; i < N; i++)
7    for (j = 0; j < M; j++)
8      data[i][j] -= mean[j];
```

(a) Initial source

```
1  for (j_tile = 0; j_tile < ceil(M/128); j_tile++) {
2    for (j = (128 * j_tile); j < min(M,(128 * j_tile)); j++) {
3      for (i = 0; i < N; i++)
4        mean[j] += data[i][j];
5      mean[j] /= float_n;
6    }
7  }
8  for (i = 0; i < N; i++)
9    for (j = 0; j < M; j++)
10     data[i][j] -= mean[j];
```

(b) Tile the for-loop with 128

```
1  for (j_tile = 0; j_tile < ceil(M/128); j_tile++) {
2    double transient_mean = new int[128]
3    for (j = (128 * j_tile); j < min(M,(128 * j_tile)); j++) {
4      for (i = 0; i < N; i++)
5        transient_mean[j] += data[i][j];
6      transient_mean[j] /= float_n;
7    }
8    memcpy(mean[128 * j_tile], transient_mean, 128 * sizeof(int));
9  }
10 for (i = 0; i < N; i++)
11   for (j = 0; j < M; j++)
12     data[i][j] -= mean[j];
```

(c) Further introduce a transient for the array *mean*

Figure 1 The *covariance* kernel from PolyBench [15], and the results of applying a sequence of two transformations

An example of a pair of cyclic transformations is shown in Figure 2, MapExpansion — a transformation that converts an N-dimensional map into N unidimensional maps, and its inverse, the MapCollapse — a transformation that converts multiple unidimensional maps into a single multi-dimensional map [17]. We address this challenge, proposing a pruning strategy (Section 3.2) that allows us to avoid the application of cyclic transformations.

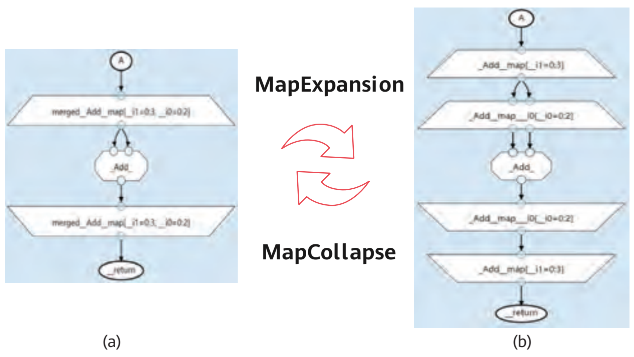


Figure 2 Example of cyclic transformations: MapExpansion and MapCollapse

3 DCTuner

DCTuner is an auto-tuning method, built around the DaCe framework [2], that explores in a greedy fashion the *deep* transformation space, leveraging a greedy search algorithm and a pruning strategy. Figure 3 shows the main components of the iterative method — (1) pruning, (2) evaluation, and (3) selection — as well as the connection points between DCTuner and the DaCe framework.

We start from a dataflow implementation of the original application expressed as a Stateful DataFlow Multi-Graph (SDFG) [2]. We input the *Original SDFG* in the DaCe framework to obtain a list of suitable transformations. Table 1 lists the DaCe's transformations used by DCTuner, the set of applicable transformations is generated by DaCe based on the characteristics of the input application, so it is input dependent. DCTuner then performs the *pruning* phase (1): using a *transformation log* — a list of already applied transformations (initially empty) — and, according to our strategy, some of the transformations matched by DaCe are discarded. The remaining transformations are then sent to the *evaluation* phase (2), executed on the target architecture, and the performance metrics of these candidate transformations are collected. Finally, in the *selection* phase (3) the best SDFG implementation is selected, it is then sent to the DaCe framework to start a new iteration, and the last applied transformation is added to the *transformation log*. The above steps are repeated until either no suitable transformation is found or the number of iterations reaches a set threshold. DCTuner outputs the global best SDFG found.

The remaining part of this section describes in detail the search algorithm (Section 3.1) and the pruning strategy (Section 3.2).

3.1 The Search Algorithm

As discussed in Section 2.1, the exploration of the implicit transformation space of an application is computationally expensive. For this reason, DCTuner uses a **greedy** search algorithm to perform its exploration, selecting and applying only the candidate transformation that is likely to generate the best result at each iteration.

Algorithm 1 shows the pseudo-code of our greedy search algorithm. The variable sg' represents the SDFG to be transformed at each iteration — the first iteration sg' is initialized with the original implementation, $best_{sg}$ and $best_{graph}$ are the global and current best SDFG, and their performance is $best\_sg_{perf}$ and $best_{perf}$, respectively. The variable app_{tf} records the history of the applied transformations, and *threshold* is set by the user to control the maximum number of iterations to be performed. The function **Generate_TF** generates all transformations matched by DaCe on the current SDFG (sg') and outputs

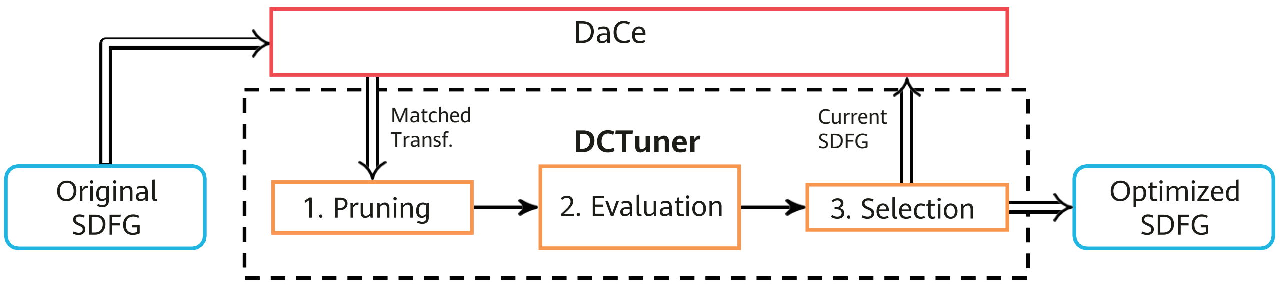


Figure 3 DCTuner method

Table 1 DaCe Transformations used by DCTuner (More information can be found in the DaCe documentation [17].)

Transformation	Explanation	Parameter
InLocalStorage	adds a transient data node between two scope entry nodes.	None
OutLocalStorage	adds a transient data node between two scope exit nodes.	None
MapCollapse	takes two nested maps with *M* and *N* dimensions respectively, and collapses them to a single *M*+*N*-dimensional map.	None
MapExpansion	takes an *N*-dimensional map and expands it to *N* unidimensional maps.	None
MapFission	replicates the map into maps in all of its internal components.	None
MapFusion	fuses the patterns MapExit -> AccessNode -> MapEntry and removes the transient in between.	None
MapInterchange	takes two nested maps and interchanges their position.	None
MergeArrays	merges duplicate arrays connected to the same scope entry.	None
RedundantArray	removes the redundant array.	None
StripMining	takes as input a map dimension and splits it into two dimensions.	tile size = 64
MapTiling	tiles in every dimension of the matched Map.	tile size = 128
InlineSDFG	inlines a single-state nested SDFG into a top-level SDFG.	None
StateFusion	takes two states that are connected through a single edge, and fuses them into one state.	None
TransientReuse	finds all possible reuses of arrays, decides for a valid combination and changes SDFG accordingly.	None

them into a list containing the candidate transformations called *candidates*. The function **Prune** takes app_{tf} and *candidates* and removes the redundant solutions according to our pruning strategy described in Section 3.2.

The function *Apply* applies the candidate transformations from the list *candidates* to *sg'*. The implementations resulting after the application of each candidate transformation are stored in the list sg_{list}. The function *Eva* takes as input list sg_{list}, outputs the current best implementation (best_{graph}), its performance (best_{perf}), and deletes other implementations to reduce algorithmic space cost. After evaluation, **to prevent our search from getting stuck in local minima**, even if $best_{graph}$ is worse than the current implementation (*sg'*), DCTuner updates *sg'* with $best_{graph}$ and continues its exploration. This allows DCTuner to explore more, possibly better, solutions. The last applied transformation is recorded into app_{tf}, and $best_{sg}$ is updated if $best_{perf}$ is better than $best\_sg_{perf}$.

The above steps are repeated until *candidates* is empty or the number of iterations is equal to *threshold*. Finally, the function **Generate_Implement** outputs the global best SDFG found ($best_{sg}$).

We further analyze the time complexity. Given an initial SDFG, assuming that the *threshold* is set to n iterations and each iteration has m candidates at most. Our search algorithm selects the current best implementation at each iteration and explores its matched transformations during the successive iteration, thus having $O(n * m)$ time complexity.

The exhaustive search has a time complexity of $O(m^n)$ as it is required to visit all candidate transformations. The complexity of a backtracking based algorithm is instead between $O(n * m)$ and $O(n^m)$.

In summary, our search algorithm employs a greedy search algorithm to achieve two key advantages:

- It can automatically perform a search in the implicit transformation space, addressing the challenges mentioned in Section 2.1 with a greedy search method and a pruning strategy.
- Compared to the exhaustive search or the backtracking algorithm, our search method enjoys lower algorithmic complexity.

Algorithm 1 DCTuner's search algorithm

```
Require: initial SDFG: sg
Ensure: optimized implementation
 1: sg', $best_{sg}$, $app_{tf}$, threshold, it, $best\_sg_{perf}$ = Init(sg)
 2: while it ≤ threshold do
 3:     candidates = Generate_TF(sg')
 4:     candidates = Prune(candidates, $app_{tf}$, threshold)
 5:     if candidates ≠ ∅ then
 6:         //Evaluate performance
 7:         $sg_{list}$ = Apply(candidates, sg')
 8:         $best_{graph}$, $best_{perf}$ = Eva($sg_{list}$)
 9:         //Update current SDFG and global best SDFG
10:         sg', $app_{tf}$ = Update($best_{graph}$)
11:         if $best_{perf} \geq best\_sg_{perf}$ then
12:             $best_{sg} \leftarrow best_{graph}$
13:         end if
14:         $it \leftarrow it + 1$
15:     else
16:         break while
17:     end if
18:     return Generate_Implement($best_{sg}$)
19: end while
```

3.2 The Pruning Strategy

A crucial part of the DCTuner method is the pruning strategy. This strategy is used to remove some of the candidate transformations matched by the DaCe framework, allowing the search algorithm to avoid *cyclic transformations* (see Section 2.1) and the re-evaluation of equivalent SDFGs. The core idea behind the pruning strategy is to remove candidate transformations, which would invert the effect of a previously applied transformation. To do this we use a *transformation log*, containing the history of all the transformations already explored, and we use a set of *pruning rules*.

Each pruning rule is composed by two parts. The first part matches transformations **already applied** to the current SDFG which are recorded in the *transformation log* — referred to as app_{tf} in Algorithm 1. The second part of the rule matches instead the transformations **to be applied** to the current SDFG, which is proposed by DaCe (these are represented in Figure 3 by the arrow labeled *Matched Transf.*). If both parts of a rule are matched, in the *transformation log* and in the transformation list proposed by DaCe, respectively, the pruning rule is triggered and the transformation that triggered the rule is discarded.

Table 2 contains some of the pruning rules used by our pruning strategy. In the **Transformation History** column there is the first part of the rule to be matched in the *transformation log*. The **Candidate to Be Pruned** column contains the second part of the rule. Intuitively, if any of the pair of transformations in Table 2 is applied in sequence, the second transformation will invert the effect of the first transformation. For example, if we take the first rule, applying the transformation *Exchange Loop B and A* would invert the effect of the already applied transformation *Exchange Loop A and B*.

The last two rules in Table 2, relative to the *tiling* and *strip* transformations are different from the others. The aim of these rules is not to avoid *cyclic transformation* but to limit the application of the tiling and strip transformations which could otherwise be indefinitely applied to the same loop nest. We found these rules to be beneficial in enhancing the search speed in most of the cases at the cost of a small performance degradation for a small set of kernels (e.g., *Matrix Multiplication*).

Figure 4 illustrates how our pruning strategy integrates with our search algorithm. The example contains the

Table 2 Example of pruning rules

Transformation History	Candidate to Be Pruned
Exchange Loop A and B	Exchange Loop B and A
Expand Loop A to A_1, A_2	Collapse Loop A_1 and A_2
Collapse Loop A, B to AB	Expand Loop AB
Fission Loop A to A_1, A_2	Fusion Loop A_1 and A_2
Fusion Loop A and B to C	Fission Loop C
Tile Loop A to A and A_1	Tile Loop A or A_1
Strip Loop A to A and Sub-Loops	Strip Loop A or Sub-Loops

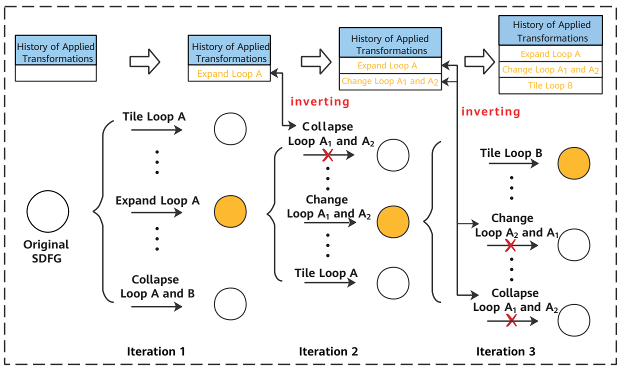


Figure 4 Application of the pruning rules

first three iterations of an auto-tuning process where the circles represent different SDFG implementations and the highlighted circles represent the best candidates found in each iteration. In iteration 1, the transformation log — containing the previously applied transformation — is empty, hence no candidates are pruned. The evaluation phase determines that the best transformation to apply in iteration 1 is **Expand Loop A**; so **Expand Loop A** is applied and recorded in the transformation log. The SDFG obtained after the application of the transformation is analyzed using DaCe, which provides a new set of transformations that could be applied in iteration 2.

The pruning phase of iteration 2, after the analysis of the *transformation log* and the current transformation candidates, triggers the second rule in Table 2 and therefore removes the **Collapse Loop A_1 and A_2** from the list of transformations to be evaluated. Finally, in the pruning phase of iteration 3, two of the candidate transformations trigger the first and second pruning rule and are then discarded.

4 Experimental Evaluations

In this section, we present three experiments to evaluate the benefits of DCTuner. Section 4.1 describes the experimental environment used to carry out the three experiments.

The first experiment, discussed in Section 4.2, aims to evaluate the greedy exploration algorithm used by DCTuner. To assess the benefit of our greedy method we compare the performance and optimization times of DCTuner against an exhaustive exploration algorithm using 10 kernels from PolyBench; our experiments show that DCTuner greedy algorithm has on average only a 2% performance decrement compared to the exhaustive search while taking a fraction of the time to perform the auto-tuning.

In Section 4.3, we test the level of portability of a real-world HPC application — the Weather Research and Forecasting (WRF) 3.4.1 [18] — optimized using DCTuner. The experiment targets two different architectures (Intel and ARM) and compares the performance before and after DCTuner's auto-tuning. Finally, in Section 4.4, we further demonstrate the use of DCTuner on another real-world HPC application, the Community Earth System Model (CESM) 2.1.1 [19], and we compare the performance against the original version, the version optimized by DaCe and the version optimized with DCTuner.

4.1 Experimental Setup

Table 3 shows the details of the architectures used in our experiments. The experiments in Section 4.2 and Section 4.4 use a Kunpeng 920 node (ARM). The experiment in Section 4.3 compares the performance of both target platforms, ARM and Intel.

Table 3 Target platforms

Platform	Hardware	Software
aarch64	998.4 GFLOPS (10.39 GFLOPS/core) NUMA nodes: 4 Memory: 512 GB L1: 128 KB, L2: 512 KB, L3: 49152 KB	CentOS 7.6 HCC Compiler OpenMPI 4.0.3 CMake 3.15.0 Python 3.7.4
x86_64	768 GFLOPS (32 GFLOPS/core) NUMA nodes: 2 Memory: 384 GB L1: 64 KB, L2: 1024 KB, L3: 36608 KB	CentOS 7.6 GCC 9.3.0 OpenMPI 4.0.3 CMake 3.15.0 Python 3.7.4

As target applications, we used the PolyBench benchmark and two real-world HPC applications: WRF and CESM. We first ran a profiling step using Perf to determine the hot-spots and focus on the optimization of the hot-spots. Because the original implementations of the WRF and CESM are in FORTRAN, to be able to use our method, we had to transform the implementation of the identified hot-spots in SDFG. In the future, we plan to automate the extraction of the SDFG from FORTRAN and C applications. The PolyBench kernels were instead entirely translated to SDFGs. For all of the experiments shown, we used the highest optimization level (i.e. O3), the tile-sizes used in **MapTiling** and **StripMining** are respectively 128 and 64 as per DaCe default parameters (see Table 1).

4.2 Evaluation of DCTuner Search Algorithm

To evaluate the goodness of DCTuner search algorithm we compare the performance gain obtained by DCTuner against the optimal combination of transformations obtained by performing an exhaustive search in the transformation space. To complete this experiment in a feasible amount of time, we first perform the auto-tuning procedure using the DCTuner auto-tuning algorithm, we record how many iterations were needed to find the global best implementation, then we perform an exhaustive search in the transformation space for the same number of iterations to compare the optimization effects (a search up to *N* iterations in this context refers to the evaluation of *N* transformations applied sequentially to the original implementation).

We ran the experiment on 10 PolyBench kernels. We selected these kernels among all PolyBench kernels as these were the only ones able to successfully conclude the exhaustive search; for the remaining kernels the transformation space was too large — containing 100,000 to 400,000 candidate transformations — to evaluate in the budgeted time. The results shown in Figure 5a, show that the execution time of the kernel optimized using DCTuner is comparable to that of the kernels optimized using exhaustive search, with an average performance degradation of only **2.16**% compared to the optimal solution found using the exhaustive search. The ADI kernel had the biggest performance gap between greedy and exhaustive search, where the implementation obtained using the greedy search was 6.96% slower than the optimal solution. Figure 5b shows the execution time of performing *T* iterations of the exhaustive search, in which *T* is the number of iterations where the optimal solution appears.

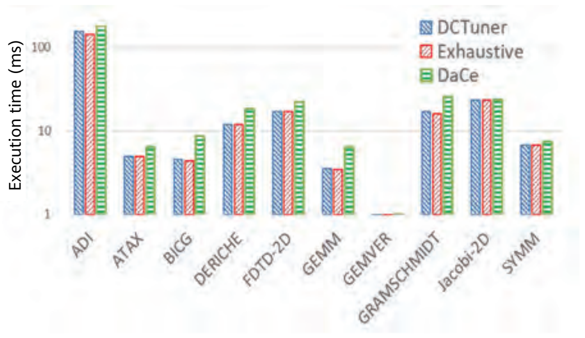
Even with this comparison, the time overhead of exhaustive search is still impractical, taking on average **30x** more time compared to our search algorithm. In the worst-case scenario (FDTD-2D), the exhaustive search took 79x more time. The Deriche kernel was the only kernel for which the exhaustive search took less time than DCTuner's; this was caused by the fact that the optimal solution appears in the first iteration, i.e., this is just the time to perform one iteration of the exhaustive search.

Additionally, we compared the performance obtained using DaCe without our auto-tuning optimization (Figure 5a). The results show that DCTuner is able to improve the performance of every kernel, with an average increase in performance of **37.67% over DaCe**. The implementation generated by DCTuner of the BICG kernel was the one that showed the biggest performance gap compared to the version generated by DaCe, with an improvement of **86.54**%.

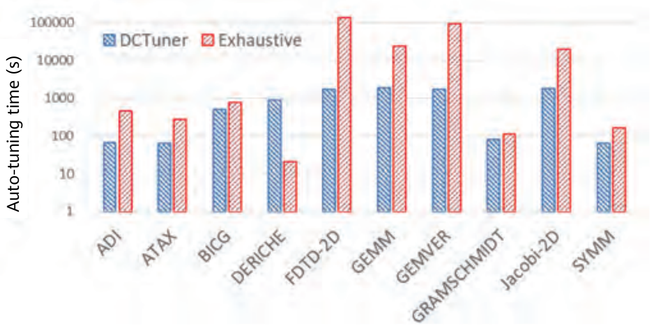
In summary, this experiment showed that the greedy search strategy used by DCTuner was able to find solutions close to optimal — with an average performance degradation of 2.16% — in a fraction of the time required by an exhaustive search — 30x less auto-tuning time on average, and that our solutions were consistently better than the solutions proposed by DaCe without auto-tuning — 37.67% better on average.

4.3 Evaluation of DCTuner Portability on WRF

To assess the performance portability of applications auto- tuned using DCTuner we measured the performance of WRF on the **ARM** architecture and on the **Intel** architecture (Table 3), with and without the DCTuner auto-tuning step. The results are shown in Figure 6. In both plots, the y-axis shows the execution time on the two architectures, while the x-axis shows the "Simulation Time", a parameter of WRF that specifies the length of the simulation expressed in hours. Figure 6a shows the results obtained on the ARM architecture; we observe that DCTuner improves the performance of the original application by 13.98% on average, and up to 15.01%. Moreover, the performance obtained by DaCe is almost equivalent to that of the original implementation, and we obtain a similar performance improvement over DaCe of **13.76%** on average, and up to 14.67%.

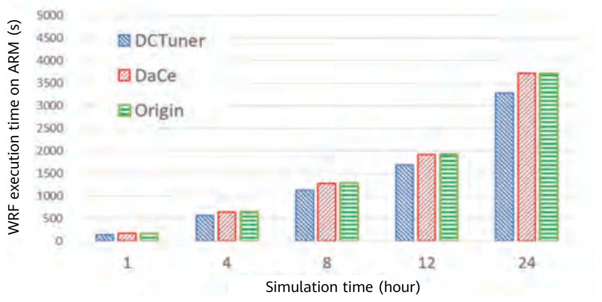


(a) Execution time (lower is better, time shown in log-scale)

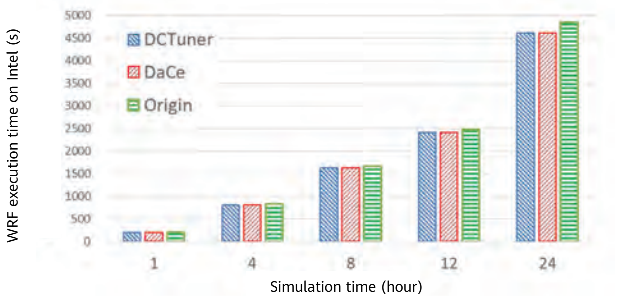


(b) Optimization time (lower is better, time shown in log-scale)

Figure 5 Comparison between DCTuner search algorithm and Exhaustive search and DaCe



(a) Execution time on **ARM** (lower is better)



(b) Execution time on **Intel** (lower is better)

Figure 6 Evaluation of performance portability, Intel vs ARM

The experiment on the Intel architecture shows us different results, as shown in Figure 6b. In this case, our auto- tuning method was still able to improve the performance of the original implementation by **3.42%** on average, but has a negligible improvement over DaCe — **0.14%** on average.

From this experiment, we conclude that DCTuner is able to improve the performance of WRF on the ARM architecture, and produces similar results on the Intel architecture if compared to DaCe. We are still investigating the data of this experiment; however, we believe that these results might be explained by an affinity of DaCe to Intel architectures and that DCTuner — through the auto-tuning process — is able to bridge the performance gap on ARM architectures.

4.4 Evaluation of DCTuner on CESM

In this experiment, we wish to show that DCTuner can fruitfully be used to optimize another real-world HPC application: CESM. We measured the performance on the ARM architecture (Table 3) and compared the implementation optimized using DCTuner against the original implementation and against DaCe without our auto-tuning method.

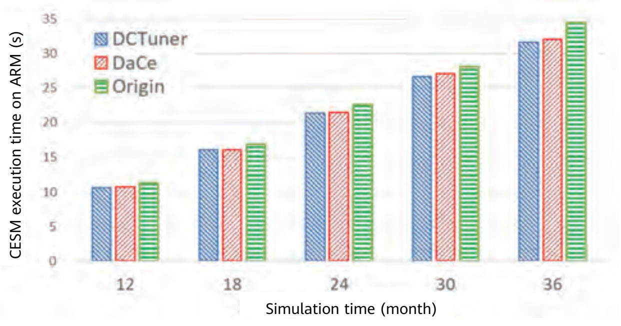


Figure 7 CESM execution time on the ARM architecture (lower is better)

Figure 7 shows the results we obtained on the CESM application using DCTuner and DaCe without auto-tuning compared to the original implementation. The execution times were measured on the ARM architecture. The results show an average performance improvement obtained using DCTuner over the original implementation of **6.15%** and minor performance improvement compared to DaCe without auto-tuning of **1.01%** on average.

In summary, these results show that our method can be applied to real-world HPC applications. We observed that the optimization performed by DaCe without our auto-tuning step generates high-quality solutions, however, in all cases, DCTuner was able to provide an additional performance gain.

5 Related Work

A data-centric representation describes the semantics of an application using a graph which is interpreted according to the dataflow model. The edges of these graphs explicitly represent data movements and the use of this information facilitates the analysis of applications and provides more opportunities for optimization. Many optimization approaches using data-centric representation have been proposed in the machine learning field. TensorFlow [7] and PyTorch [20] directly apply all matched transformations to the original graph. TVM [9] has two optimization levels: using the graph-level it performs graph transformations, using the operator-level it auto-tunes the program parameters. MetaFlow [8] and TASO [10] deeply explore possible transformations to find an optimized graph implementation. However, these works are only suitable for computations and data layouts used in deep learning. Some studies use data-centric representation to describe HPC applications. Kodukla et al. use such a representation to improve the data locality [21]. Bamboo [11] introduces a programming language and a compiler that uses three transformations to optimize applications for many-core implementations. MODESTO [12] performs exhaustive exploration of the transformation space, but it only targets stencil kernels and it does not perform deep optimization. Our work integrates and extends DaCe [2] adding auto-tuning capabilities and addressing the challenges described in Chapter 2. DaCe is a framework that expresses applications as an SDFG using a domain-specific language. DaCe can automatically apply a set of transformations (called *strict transformations*), however, it also provides a manual process for users to perform deep optimization. Recently, DaCe was applied to an extreme-scale quantum transport simulation application in [4] and obtained excellent results.

5.1 Program Auto-tuning

The idea behind program auto-tuning is to explore a parameter space and to find a (close-to) optimal parameter combination. Normally, the parameter space is explicitly defined by the user (for example compilation options, tile size, vector length, and their ranges) [14, 22–25]. Many general-purpose methodologies have been studied, for instance, OpenTuner [14]: a mature auto-tuning architecture that employs many algorithms to search for the best solution in parallel. There are several core differences between our method and OpenTuner. Our method performs (fully) automatically program transformations. The way we achieve this is by leveraging a data-centric code representation and the DaCe framework to find areas of the application that can be optimized automatically. In OpenTuner, a user needs to instrument the application to expose optimization knobs and the user also needs to determine (statically) the set of parameters to be explored. This means that the space explored by OpenTuner is determined statically by the user. On the other hand, the space explored by DCTuner is determined automatically from the characteristic of the applications, and dynamically, as after each transformation we analyze the obtained implementation to find a new set of applicable transformations. These generic technologies are not restricted to specified applications or architectures, and effectively save laborious manual effort. Domain-specific and architecture-specific auto-tuners are also gaining traction. There are, for example, several works focusing on the optimization of stencil computation [24, 26, 27]. Other works focus on the auto-tuning of linear algebra programs [28, 29]. There are auto-tuners specific for the Fast Fourier Transformation [30, 31]. Finally, some auto-tuners specialize in tensor computation [32, 33]. Some methodologies use architecture features to auto-tune applications, and these have been applied on GPUs [34–36], FPGAs [36, 37], and NUMA architectures [38, 39].

To our best knowledge, the existing program auto-tuning methods do not perform deep optimization on data-centric representations. Performing deep optimization on data-centric representation is challenging (see Chapter 2). The auto-tuner needs to explore an implicit transformation space — which is computationally expensive, and avoid the application of inverting transformations.

6 Conclusion and Future Work

This work focuses on performance and portability improvement of high-performance computing applications using the auto-tuning methodology. We presented DCTuner, an auto-tuning method that integrates with the DaCe framework and enables deep optimization of data-centric applications. To this end, we introduced a greedy exploration algorithm that we use to explore the space of application transformations — each element of such a space represents the iterative application of multiple transformations to the original application — and a pruning strategy that avoids the evaluation of equivalent sequences of transformations.

We performed three experiments to evaluate our method. In the first experiment, we demonstrated the goodness of our greedy search strategy by using 10 kernels selected from PolyBench. We showed that the best solution found by DCTuner performs on average 2% worse than the optimal combination of transformations — determined through exhaustive search of the solution space, while being up to 30 times faster than the exhaustive search. We also showed that auto-tuning kernels with DCTuner were able to achieve up to 37% additional performance improvement over kernels optimized using DaCe. In a second experiment, we used a real-world HPC application (WRF) to evaluate the performance portability of applications auto-tuned with DCTuner. We compared the performance benefit on two target architectures, ARM and Intel. The experimental data shows that by using our auto-tuning method, we were able to bridge a performance gap between Intel and ARM, obtaining over 13% performance improvement on ARM architecture. Finally, in our last experiment, we tested our method using another real-world HPC application (CESM). Our results showed that using DaCe to optimize the applications provides the main performance improvement — above 5%. However, using our auto-tuning method we were able to gain an additional improvement of 1%.

In the near future, we plan to evaluate various performance metrics to replace or complement our evaluation phase, aiming to increase the overall auto-tuning speed of DCTuner. Additionally, we wish to automate the extraction of an SDFG representation from an application, that is currently extracted manually, in order to obtain a fully automated procedure.

References

[1] D. Padua, *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.

[2] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[3] C. L. Mendes, B. Bode, G. H. Bauer, J. Enos, C. Beldica, and W. T. Kramer, "Deploying a large petascale system: The blue waters experience," *Procedia Computer Science*, vol. 29, pp. 198–209, 2014.

[4] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–13.

[5] R. Mijaković, M. Firbach, and M. Gerndt, "An architecture for flexible auto-tuning: The periscope tuning framework 2.0," in *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*. IEEE, 2016, pp. 1–9.

[6] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás, "Trends in data locality abstractions for hpc systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, 2017.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium*

on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.

[8] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing dnn computation with relaxed graph substitutions," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 27–39, 2019.

[9] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, vol. 11, p. 20, 2018.

[10] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[11] J. Zhou and B. Demsky, "Bamboo: a data-centric, object-oriented approach to many-core software," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 388–399.

[12] T. Gysi, T. Grosser, and T. Hoefler, "Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 177–186.

[13] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Supporting stateful tasks in a dataflow graph," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 435–436.

[14] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[15] L. N. Pouchet., "Polybench: The polyhedral benchmark suite." https://sourceforge.net/projects/polybench, 2016.

[16] R. E. Korf, "Linear-time disk-based implicit graph search," *J. ACM*, vol. 55, no. 6, Dec. 2008. [Online]. Available: https://doi.org/10.1145/1455248.1455250

[17] "Dace documentation 0.10.0a," 2020. [Online]. Available: https://spcldace.readthedocs.io/en/latest/source/dace.html W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker,

[18] M. G. Duda, X.-Y. Huang, W. Wang, and J. G. Powers, "G.: A description of the advanced research wrf version 3," in *NCAR Tech. Note NCAR/TN-475+ STR*. Citeseer, 2008.

[19] "The community earth system model (cesm) large ensemble project: A community resource for studying climate change in the presence of internal climate variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333–1350, 2015. [Online]. Available: http://www.jstor.org/stable/26219645

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[21] I. Kodukula and K. Pingali, "Data-centric transformations for locality enhancement," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 319–364, 2001.

[22] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, "Cobayn: Compiler autotuning framework using bayesian networks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 2, pp. 1–25, 2016.

[23] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 307–316.

[24] T. Lutz, C. Fensch, and M. Cole, "Partans: An autotuning framework for stencil computation on multi-gpu systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–24, 2013.

[25] Y. Zhu and J. Liu, "Classytune: A performance auto-tuner for systems in the cloud," *IEEE Transactions on Cloud Computing*, 2019.

[26] Y. Luo, G. Tan, Z. Mo, and N. Sun, "Fast: A fast stencil autotuning framework based on an optimal-solution space model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 187–196.

[27] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 676–687.

[28] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[29] A. Rasch, R. Schulze, and S. Gorlatch, "Generating portable high-performance code via multi-dimensional homomorphisms," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 354–369.

[30] Y. Li, Y.-Q. Zhang, Y.-Q. Liu, G.-P. Long, and H.-P. Jia, "MPFFT: an auto-tuning FFT library for OpenCL GPUs," *Journal of Computer Science and Technology*, vol. 28, no. 1, pp. 90–105, 2013.

[31] Z. Li, H. Jia, Y. Zhang, T. Chen, L. Yuan, L. Cao, and X. Wang, "AutoFFT: a template-based FFT codes auto-generation framework for ARM and X86 CPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–15.

[32] A. Mazaheri, J. Schulte, M. W. Moskewicz, F. Wolf, and A. Jannesari, "Enhancing the programmability and performance portability of GPU tensor operations," in *European Conference on Parallel Processing*. Springer, 2019, pp. 213–226.

[33] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on GPUs," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 85–95.

[34] W. Jia, E. Garza, K. A. Shaw, and M. Martonosi, "Gpu performance and power tuning using regression trees," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 2, pp. 1–26, 2015.

[35] M. Steuwer, T. Remmelg, and C. Dubach, "Matrix multiplication beyond auto-tuning: rewrite-based gpu code generation," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2016, pp. 1–10.

[36] M. Tillmann, T. Karcher, C. Dachsbacher, and W. F. Tichy, "Application-independent autotuning for gpus." in *PARCO*, 2013, pp. 626–635.

[37] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning FPGA compilation," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 157–166.

[38] M. Eljammaly, L. Karlsson, and B. Kågström, "An auto-tuning framework for a numa-aware hessenberg reduction algorithm," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 5–8.

[39] M. Popov, A. Jimborean, and D. Black-Schaffer, "Efficient thread/page/-parallelism autotuning for NUMA systems," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 342–353.

Humble Heroes

A. N. Yzelman

Abstract

The concept of a *humble programmer*, coined by Dijkstra in the 1970s, fully acknowledges the complexity of programming, deeming it unsuitable for the human mind. This concept leads to the design of humble programming frameworks that not only simplify the implementation of algorithms, but also automate the optimization and scalability of algorithms across compute units. Such humble frameworks typically favor productivity over performance.

By contrast, *hero programmers* are experts with in-depth knowledge of parallel algorithms, applicable lower bounds, coding, and hardware, typically seeking to achieve the highest possible efficiency on any given system. Given that hardware complexity and diversity are set to increase in the years to come, *humble programmers* will not be able to drive increasingly more complex and larger-scale systems using contemporary software technologies. This, combined with the scarcity of hero programmers at the present time and in the foreseeable future, highlights the need for humble programming models that can achieve good performance.

This paper starts by outlining the successful humble MapReduce and Pregel vertex-centric frameworks and identifies the common design factors that they share with the novel *Algebraic Programming (ALP)* paradigm. Satisfying the need for supporting multiple humble programming models while reducing demands on hero programmers, this paper proposes a vertex-centric programming model on top of ALP, demonstrating that multiple humble programming models can be supported by a single software stack. Thus, fewer heroes may support a greater number of humble programmers.

Experiments demonstrate that the resulting ALP/Pregel paradigm scales well on a shared-memory parallel system, achieving speedups of up to 17.8x on common graph workloads. Even though vertex-centric algorithms that solve a given problem commonly differ from their canonical solutions, this paper compares such an algorithm (used for ranking web pages) with a highly optimized canonical solution for completing the same task. The result shows that in sequential execution ALP/Pregel achieves up to a 8.99x speedup, outperforming the canonical solution in 12 out of 13 datasets tested. In shared-memory parallel execution, ALP/Pregel achieves a 0.276–5.76 times speedup over a highly optimized implementation of the canonical PageRank algorithm. The slowdowns, which are down to 0.276x, are mostly due to the workspace not fitting in L3 cache for the vertex-centric algorithm. For the smallest and largest datasets where such cache effects are less apparent, ALP/Pregel is faster in 3 out of 6 instances with speedups of up to 5.73x. This indicates that humble algorithms may still achieve hero performance, contributing a solution to the looming crisis in software productivity.

1 Introduction

The perhaps esoteric title of "Humble Heroes" relates to one of the major challenges in computing: programming potentially highly parallel and highly heterogeneous resources, without complicating the task to the point where only a small percentage of highly skilled programmers can write efficient programs.

Writing software grows more and more difficult due to the increasing complexity of computer architectures, coupled with a diverse and growing range of architectures that programs should cope with. The recent hardware trends show increase both in the number of non-uniform memory access (NUMA) designs and the degree of non-uniformity — there are already four to eight NUMA domains on a modern CPU — while both the per-core memory capacity and bandwidth are decreasing. These factors will significantly increase pressure on future software design for any given architecture. Additionally, heterogeneity may appear on a wide spectrum from a single chip consisting of different types of processing units to large-scale data centers connecting a diverse range of architectures. Within and between these extremes, heterogeneity will realistically continue to appear.

With the diversification of architectures and the increase of their design complexity and heterogeneity, contemporary software technologies no longer scale. Maintaining standard libraries, for example, requires dedicated programmers with theoretical, algorithmic, domain, programming, and hardware expertise in order to achieve acceptable performance on a given architecture. The required breadth of expertise for maintaining standard libraries sets this class of programmers apart from the majority — colloquially, we call them *hero programmers*.

1.1 Humble Programming

In contrast to hero programmers, Dijkstra first coined the concept of a *humble programmer*. Paraphrasing his monologue [Dij72], this concept entails *approaching programming while fully recognizing the incredible complexity of the task, by preferring modest and elegant programming languages that respect the limitations of the human mind*. Perhaps implicit in humble programming, there is acceptance of a performance trade-off, favoring programming productivity over performance. This paper, however, provides evidence that performance and productivity need not be mutually exclusive.

Contemporary examples of humble programming models that scale across large-scale computational resources include MapReduce [DG08], Pregel [MAB+10], and Spark [ZCF+10], all of which make it possible to drive complex large-scale systems using simple, elegant, and easy-to-use abstractions that any programmer can understand. These programming models have been extremely successful in providing parallel compute power to humble programmers, allowing them to quickly deploy and scale up a variety of workloads.

1.2 The Software Bottleneck

Hero programmers, by contrast, strive to achieve peak performance — or as close to it as possible — for any given workload on any given architecture. They embrace the complexity that comes with the task, using low-level interfaces such as assembly language, intrinsics, threading interfaces, remote direct memory access, and message passing. The resulting code requires significant time and effort to build and is costly to maintain. Updating the code to novel versions of the architecture it was originally written for remains costly, while supporting completely novel architectures requires a ground-up duplication of all these efforts.

If the future is to be increasingly heterogeneous in nature, then the future software stack must not be dependent on hero programmers: there will never be enough expert programmers to support the many application domains that require mapping to different architectures. Instead, the future software stack must speak primarily to humble programmers, be deployable on different architectures, and achieve good performance — all while avoiding the software bottleneck.

1.3 Outline

Section 2 reviews the MapReduce and the vertex-centric Pregel [MAB+10] programming interfaces, providing a more in-depth characterization of the properties of successful humble programming models. Section 3 describes the novel *Algebraic Programming (ALP)* humble programming paradigm, and in particular ALP/GraphBLAS, which achieves hero-level performance on both shared- and distributed-memory systems [YDNNS20].

Section 4 notes that, while ALP may be a significant leap forward for anyone comfortable with using algebraic annotations to express algorithms, some humble programmers may prefer to use alternative paradigms. The section asserts that neither forcing a single paradigm on humble programmers nor supporting an unchecked set of potentially many humble paradigms solves the challenge of productively using future large-scale heterogeneous systems. Instead, it advocates the alternative approach this paper takes: automatic translation of one humble programming model into another.

Section 5 realizes this ideal for Pregel and ALP, by introducing a C++ API for translating vertex-centric programming into ALP at compile time, thus supporting multiple humble paradigms with a single software stack. Section 6 presents experimental results, showing that the cost of simulating a Pregel interface on top of ALP is negligible and that such an interface achieves comparable performance to direct programming in ALP/GraphBLAS — it even outperforms direct programming if some relaxations are permissible. Section 7 concludes this paper and presents a future outlook.

2 Influential Humble Programming Models

This section discusses two pivotal humble programming models that have made massively parallel computing accessible to the public rather than to a limited group of parallel programming experts: MapReduce and Pregel.

2.1 MapReduce

In MapReduce, data items are key-value pairs (k, v) from domains K and V, respectively, and programs are a sequence of two alternating phases: *map* and *reduce*.

Let $D_0 \subset K \times V$ be the set of initial data items. The i-th *map* phase takes each key-value pair d_k from D_{2i}, where the integer k is in the range of $[0, |D_{2i}|)$, and maps it to a possibly empty set of new pairs $f_i(d_k)$. Then, one of the following may happen: the original pair d_k is filtered out, one new pair is generated, or an arbitrary number of new pairs is generated. The mapping function $f_i : K \times V \to \mathcal{P}(K \times V)$ is user-defined and may be different for each round i. The original set of data D_{2i} is discarded and replaced by the set of transformed entries D_{2i+1}.

The i-th *reduce* phase operates on all keys in the current data set D_{2i+1}, specifically, the set $K_i = \{k \in K \text{ s.t. } \exists(k, v) \in D_{2i+1}\}$. For a key k, assume there are $r > 0$ key-value pairs $(k, v_0), \ldots, (k, v_{r-1})$, then the reduction operation computes $v' = \odot_{i=0}^{r-1} v_i$ and produces an entry (k, v') in $D_{2(i+1)}$. The entry replaces all key-value pairs with key k in D_{2i+1} — this repeats for each key in K_i. As with the user-defined mapping function, the reduction operator $\odot$ is user-defined and may differ for each round.

While the mapping operations may be arbitrary, it is helpful if the reduction operators are associative, in which case the reduction can be automatically parallelized by systems like ALP; otherwise, the *reduce* phase can only be parallelized over each key $k \in K_i$, limiting the amount of parallelism exposed.

The round-based nature of a MapReduce communication, which consists of the *map* phase and the *reduce* phase, is closely related to the bulk synchronous parallel (BSP) paradigm of parallel computing [Val90]. The *map* phase induces no communication between parallel compute units whereas the *reduce* phase does require communication. BSP models a parallel system as local compute units with local memory, interconnected by a full-duplex network. Each local compute unit executes a program in two rounds — like MapReduce does. Each round includes two phases: 1) local computations using local memory elements, and 2) arbitrary data movement patterns between memory elements. On any one compute unit, a next round may only proceed when all incoming and outgoing messages have been received and sent, respectively. This does not imply that a global synchronization barrier exists between rounds.

There are two differences between direct-mode BSP and MapReduce: 1) BSP operates at the coarse granularity of one task per process while MapReduce operates at the granularity of the dataset sizes $|D_i|$; 2) BSP allows arbitrary communication patterns while MapReduce always sorts on keys. L. G. Valiant [Val90], however, proposed an *automatic mode* of BSP that allows the simulation of PRAM algorithms on a BSP architecture. The hashing mechanism employed in this mode is typically mirrored in MapReduce implementations such as Hadoop [SKRC10], in Pregel realizations such as Giraph [Ave11], and in the Spark framework [ZCF+10].

Such simulation techniques come at a cost and preclude the application of communication-optimal algorithms. Thus, orders of magnitude performance differences between humble programming frameworks and the direct-mode BSP are not uncommon. Suijlen and Yzelman made a performance comparison between ALP/GraphBLAS and Spark [SY19] (when both use ten nodes) and reported a two-orders of magnitude performance difference for the PageRank algorithm on a moderate data size (of approximately 298 million edges).

2.2 Pregel

Pregel is a programming model for graph computations [MAB+10]. Assume the data graph $G = (V, E)$, then Pregel allows defining a round-based computation for each vertex $v \in V$ to execute. Each round consists of two steps: 1) execution of the user-defined program on each vertex; and 2) exchange of messages via the edges E. Messages are broadcast from vertices $v \in V$, that is, a message $m \in M$ is sent to all neighbors $N(v) = \{w \in V \mid (v, w) \in E\}$ of v.

Such *vertex-centric* programming models have achieved great success and inspired a significant number of vertex- and edge-centric programming frameworks [MWM15]. The round-based nature of vertex-centric programming, like that of MapReduce, is also a variation of the BSP model — like MapReduce, vertex-centric programming applies parallelism in a fine-grained fashion by mapping the programs on the input data itself. Most variants of Pregel follow the principle that while the algorithm is strictly round-based, the execution of such a program need not be. As such, latency may be hidden by overlapping the message communication phase of one set of vertices with the computational phase of other sets of vertices. This resonates with Valiant's automatic-mode BSP, which performs similar latency hiding through overlapping communication and computation phases [Val90].

Consider, as an example of a vertex-centric program, the problem of determining the strongly connected components (SCCs) of an undirected graph $G = (V, E)$, that is, identifying the minimum number of subsets $V_k \in V$ where (V_i, E), are strongly connected[1]. A simple vertex-centric program determines these subsets in two stages: 1) Each $v \in V$ is assigned a unique identifier; 2) During each round, each vertex sends its current identifier to its neighboring vertices, and overwrites the current identifier to the largest value contained in all incoming messages if that value is strictly larger than the current identifier. If the current identifier is not overwritten in the second stage, the vertex votes to halt the program. When all vertex programs vote to halt the program, the execution terminates. Once the execution terminates, the number of SCCs k corresponds to the number of unique identifiers that remain, and the component to which each vertex belongs is indicated by its identifier.

[1] A graph or subgraph is strongly connected only if there is a path between any two of its vertices.

This vertex-centric algorithm and its termination criterion are intuitive to understand and therefore fit the bill of a humble solution for solving the SCC problem. However, if the input to this algorithm is an undirected line graph, the number of rounds required will be $|V|$, which means that $|V|$ vertex programs will be active per round, and that the work for this SCC algorithm amounts to $\Omega(|V|^2)$. By contrast, the Awerbuch-Shiloach [SV80, AS87] algorithm for SCC can achieve $\mathcal{O}(|E| \log |V|)$ work complexity. Algorithms with such improved bounds can be implemented on top of a vertex-centric framework but will be far less humble in nature [SW14]. The same holds for ALP and GraphBLAS; consider, for example, the linear-algebraic variant of Awerbuch-Shiloach by Zhang et al. [ZAB20].

Since adversarial graph structures such as line graphs do not naturally occur in the real world, the quadratic algorithm may be acceptable in practice — in which case the humble choice suffices. This paper, in line with its motivation, thus considers only easy-to-understand humble algorithms, leaving the important task of finding and realizing asymptotically optimal algorithms to the hero programmers.

2.3 Common Factors

The concepts shared by the successful MapReduce and Pregel programming models are as follows: First, they both operate on sets of data and express parallelism mainly by operating concurrently on data elements within those sets, which is known as a *data-centric* approach. Second, both their ways of expressing parallelism are implicit. Programmers express operations by mutating one set of data into another without considering their parallelization. Third, operations proceed round-by-round, thus exposing a sequential view of the final programs. This is similar to the direct-mode of BSP and traditional imperative programming. Fourth, efficient implementations of these two models exploit the imposed program structure to enable both scalable execution and a high degree of overlap between executing rounds and phases — this is achieved by overlapping communication with computation and exploiting parallel slackness [Val90, MAB+10]. These four concepts are furthermore also shared with another successful humble programming model: Spark [ZCF+10].

3 Algebraic Programming

The ALP paradigm provides a data-centric, sequential, and imperative programming approach based on algebraic concepts and annotations. Designed to be humble, this paradigm presents three algebraic concepts to programmers: containers, structures, and primitives. ALP/GraphBLAS, introduced by Yzelman et al. as a C++ realization of the GraphBLAS [YDNNS20, BMM+17, BBM+19], focuses on sparse linear algebra. In ALP/GraphBLAS, containers take the form of vectors and matrices; structures can be binary operators, monoids, or semirings; and operations can be element-level applications of a binary operator, a matrix–matrix multiplication, and so on.

3.1 Algebraic Containers

Containers in ALP, when declared, are initially *empty* — meaning that they contain no values. By default, the minimum capacity of a container equals its maximum dimension. That is, a newly created vector of size n holds no values but has the capacity for n values. Similarly, a novel $m \times n$ matrix holds no values, and its default capacity is at least $\max\{m, n\}$. Larger or smaller capacities may be set when new containers are created, while existing containers can be resized to hold more or fewer values through `grb::resize`. An ALP implementation may assign capacities that are larger than requested, but not smaller. If a requested minimum capacity cannot be guaranteed, an error will be returned.

Using standard template library (STL) compatible iterators to C++ STL containers, data can be *ingested* into ALP containers via the two primitives: `grb::buildVectorUnique` and `grb::buildMatrixUnique`. The 'Unique' suffix in these primitives indicates that a source container shall not contain duplicate entries, i.e., multiple values shall not map the same container coordinate. The current size of a vector is returned via `grb::size`, the row size and column size of a matrix are returned via `grb::nrows` and `grb::ncols`, respectively, and the current number of values in a container can be referenced via `grb::nnz`. All values can be erased via `grb::clear`.

The following three statements are examples of creating a vector with the default capacity, a vector with potentially smaller capacity, and a matrix with the exact initial capacity as returned by a matrix file parser, respectively:

```
grb::Vector< double > x( n );
grb::Vector< bool > s( n, 1 );
grb::Matrix< void > A( m, n, parser.nz() );
```

Note that the element type of an ALP container appears as a template argument, as with the STL.

The extraction of data from ALP containers proceeds using iterators that can be retrieved through the `begin`, `cbegin`, `end`, and `cend` functions. These output iterators also conform to the STL, but support only the **const**–variants, that is, values in containers cannot be adapted through iterators. In addition to data ingestion through iterators, programmers may also set a vector to be a dense one with all its values set to a specific given value:

```
grb::set( x, 1.0 );
```

Likewise, one may set a single element of a container:

```
grb::setElement( s, true, n / 2 );
```

In the preceding examples, x corresponds to a dense vector $(1, 1, ..., 1)$, while s corresponds to a sparse vector where only one nonzero exists at position $n/2$ with the value (*true*).

3.2 Algebraic Structures and Algebraic Type Traits

An example of a binary operator is addition of double-precision floating point numbers, which ALP exposes as a C++ class template:

```
grb::operators::add< double >
```

An operator $\odot$ may have algebraic properties such as associativity $(a \odot (b \odot c) = (a \odot b) \odot c)$, commutativity $(a \odot b = b \odot a)$, and idempotency $(a \odot a = a)$. ALP/GraphBLAS exposes such properties through *algebraic type traits*, for example,

- `grb::is_associative< grb::operators::add< double > >::value`, which reads **true**;
- `grb::is_commutative< grb::operators::divide< float > >::value`, which reads **false**; or
- `grb::is_idempotent< grb:operators::min< unsigned int > >::value`, which reads **true**.

Algebraic type traits can be inspected at compile-time, thus enabling semantic checks and compile-time optimizations guided by algebraic properties.

Richer algebraic structures include monoids and semirings, which may be *composed* of operators and identities. For example,

```
grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >
    grb::identities::zero, grb::identities::one
>
```

describes the standard numerical addition and multiplication over doubles, also called the *plus-times* semiring. Operators each have a domain name attached as a template argument. Identities must have an element in those domains and otherwise the code will not compile.

For any binary operator $\odot : D_1 \times D_2 \to D_3$, the three domains potentially differ. Take a look at the following example:

```
typedef grb::operators::argmax<
    std::pair< size_t, float >,
    std::pair< size_t, float >,
    size_t
> ArgmaxUINT_FP32;
```

This example describes the argmax operator over tuples of integers and floating point numbers, resulting in an integer as per

$$argmax((i_0, \alpha_0), (i_1, \alpha_0)) = \begin{cases} i_0, \text{ if } \alpha_1 \leq \alpha_0 \\ i_1, \text{ otherwise.} \end{cases}$$

Such binary operators may still form monoids or semirings, for example,

```
grb::Monoid< ArgmaxUINT_FP32,
    grb::identities::infinity >
```

depicts a valid monoid where "infinity" over unsigned integers will be interpreted as the maximum representable value *maxint*, while "infinity" over a tuple is composed by recursion over the tuple types — which, in this case, resolves to $(maxint, \infty)$.

3.3 Algebraic Primitives

An algebraic primitive combines containers and structures, modifying the former in a way that depends on the latter. Perhaps the most simplistic operation takes two input vectors and generates one output vector by applying a given binary operator in an element-wise fashion. For example,

```
grb::Vector< double > y( n );
grb::eWiseApply( y, x, x,
    grb::operators::add< double >() );
```

computes $y = x \odot x$, where $\odot$ is given by the algebraic structure provided (simple numerical addition, in this example). Using the earlier examples that defined and populated x and after executing the above, y reads $(2, 2, \ldots 2)$.

Consider element-wise application in the sparse case:

```
grb::Vector< double > d( n, 1 );
grb::operators::assign_left_if< double, bool, double >
myOp;
grb::eWiseApply( d, x, s, myOp );
```

While x is dense, s contains only a single nonzero. Because the algebraic structure of a simple binary operator does not allow for the interpretation of missing values from a container, ALP will only apply the requested binary operation on nonzeroes x_i and s_i that appear on the same coordinate i, ignoring any values in x that do not have a matching value in s and vice versa. Consequently, the above results in a single entry, namely, $d_{n/2} = x_{n/2}$.

An associative operator, joined with an identity, forms a monoid, which allows algebraic primitives to interpret missing values in sparse containers. The following example shows the behaviors of numerical multiplication as an operator rather than a monoid, under element-wise application:

```
grb::Vector< double > oneTwo( n, 1 );
grb::setElement( oneTwo, 2.0, n/2 );
grb::operations::mul< double > mulOp;
grb::Monoid<
    grb::operations::mul< double >,
    grb::identities::one
> mulMon;
grb::eWiseApply( y, x, oneTwo, mulOp );
// y = oneTwo
grb::eWiseApply( y, x, oneTwo, mulMon );
// y = (1,..., 1,2,1,...1)
```

The `grb::eWiseApply` is an out-of-place primitive. The `grb::foldl` and `grb::foldr` provide in-place variants instead:

```
grb::foldl( y, oneTwo, mulMon );
// y = (1,..., 1,4,1,...,1 )
```

Some operations require richer algebraic structures. Consider, for example, the sparse matrix–vector (SpMV) multiplication $y = Ax$:

```
grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >,
    grb::identities::zero, grb::identities::one
> mySemiring;
grb::clear( y );
grb::mxv( y, A, x, mySemiring );
```

ALP defines that only `grb::set` and `grb::eWiseApply` can operate in out-of-place fashion, whereas all other primitives have in-place semantics. For example, the above `grb::mxv` sets y to $y \oplus Ax$ — with $\oplus$ being the additive operator of the given semiring — and does not set y to Ax. If the latter is intended, the output container must be cleared first, as in the above example.

All primitives with container output support masking. Since the vector s evaluates **true** only at position $n/2$,

```
grb::mxv( y, s, A, x, mySemiring );
```

requests only the computation of $y_{n/2}$, leaving any other elements as-is. We may also invert the effective mask through *descriptors*, which provide mechanisms that modify the interpretation of input containers such as masks, and will prove useful later in this paper. The following example updates all entries of y *except* the one at position $y_{n/2}$:

```
grb::mxv< grb::descriptors::invert_mask >( y, s, A, x,
mySemiring );
```

3.4 Element-wise Lambdas

Yzelman et al. recognized that in the *blocking* mode of execution where every primitive call must complete before returning [BMM+17], performance may suffer due to unnecessary data movement in memory-bound computations. Take the following code for example.

```
grb::operators::min< double > minOp;
grb::eWiseApply( y, x, x, minOp );
grb::eWiseApply( x, y, y, addOp );
```

If the two primitives are executed one by one, elements from x and y are brought from the main memory to the CPU core(s) twice[2]. However, this could be reduced to once if the two calls were fused. Hence, Yzelman et al. introduced the `grb::eWiseLambda` primitive that allows the execution of arbitrary lambda functions on one or more vector containers. The above snippet, for example, could be replaced with a semantical equivalent as below:

```
grb::operators::min< double > minOp;
grb::eWiseLambda( [&x, &y] (const size_t i) {
        grb::apply( y[ i ], x[ i ], x[ i ], minOp );
        grb::apply( x[ i ], y[ i ], y[ i ], addOp );
    }, x, y
);
```

This fuses the calls to y and x and may complete up to $2\times$ faster than the preceding code.

The square bracket vector operator (e.g., $x[\ i\ ]$) in ALP is only valid when 1) it is used inside a lambda function passed to `grb::eWiseLambda`, and 2) index i refers to a nonzero value that was already present in the related container when the eWiseLambda was invoked. Any other use invites undefined behavior. The first vector argument to `grb::eWiseLambda` (x on the second-to-last line) defines which indices the eWiseLambda shall iterate over. The other vector arguments (such as y on that same line) must correspond to any vectors that are captured in the lambda. For example, the following snippet will only set $x_{n/2}$ equal to 3.14, while leaving any other non-zero value of x unmodified:

```
grb::eWiseLambda( [&x] (const size_t i) {
        x[ i ] = 3.14;
    }, s, x
);
```

This element-wise lambda, apart from its intended use case of manually fusing ALP operations, provides critical functionality for translating vertex-centric programs into ALP (demonstrated in Section 5.4). Moreover, in the context of humble programming, recent work by Mastoras et al. provides mechanisms by which ALP/GraphBLAS may automatically fuse operations [MAY23, MAY22] — Section 7.2 discusses the implications of this recent development for vertex-centric programs.

3.5 Final Remarks

All ALP primitives return error codes, which this paper does not discuss for the sake of brevity. A special note, however, is that the program must guarantee sufficient capacities in output containers, as otherwise the related operation may fail. For example,

```
grb::set( s, false );
```

may fail because the requested capacity of s during construction was 1, which is smaller than its total size n. This work assumes, and in implementation ensures, sufficient vector capacities.

While this section has not introduced the ALP/GraphBLAS API or other ALP extensions in full, it lays the foundation for the remainder of this paper. For full details of the ALP/GraphBLAS API and other ALP extensions, read the papers [BMM+17, YDNNS20, MAY23, MAY22].

4 Motivation

The ALP paradigm expresses programs as sequential, data-centric, and standard C++ programs. It also automatically manages performance and takes care of the corresponding code optimizations and parallelization. For example, ALP enables distributed-memory parallel computations that

[2] This assumes a large enough n compared to the last-level cache of the target architecture.

process graphs of up to 42.5 billion edges over multiple multi-socket nodes using simple humble algorithms. This is made possible by the basic algebraic concepts with which ALP programmers annotate their programs. ALP makes use of those high-level annotations to select the appropriate optimizations automatically, allowing programmers to focus entirely on the mathematics [YDNNS20].

However, while most programmers have studied linear algebra at some point during their studies, most programmers do not consciously use linear algebraic concepts — such as monoids and semirings — on a daily basis. This limits the humble appeal of ALP. Separately, a key question is how many practical workloads naturally map to the ALP paradigm.

In motivating the broader use of ALP and other humble programming paradigms, this section identifies three broad approaches:

- educate programmers on the use of algebraic concepts in programming;
- extend ALP functionalities to support broader ranges of workloads; and
- expand other humble programming models into ALP.

This paper primarily focuses on the third approach.

4.1 Educate

No programming model or language has been successful without educating programmers on its use. In the case of ALP/GraphBLAS, education may be relatively simple because most programmers already make implicit use of algebraic concepts, for example, linear algebra and its implicit use of the real semiring, or set algebraic concepts like orders implicitly used when sorting. Moreover, the idea that programming should closely follow mathematical concepts is not new; it forms the basis behind the design of the STL and generic programming [DS00, SM09, SR14] — standard tools and languages used by a significant portion of programmers. Algebraic concepts also appear in standard computer science texts, with the earliest references to the explicit use of semirings in programming included in the seminal works by Aho, Hopcroft, and Ullman [AHU74] and Cormen, Leiserson, and Rivest [CLR92].

4.2 Extend

Ongoing research should push the boundaries of ALP applicability beyond linear algebra, graph computing, and big data [KG11, KJ18]. This paper already makes use of extensions over the C GraphBLAS standard [BMM+17, BBM+19] introduced by Yzelman et al. [YDNNS20], while Section 7.3 proposes two other extensions that would enable automatic translation of MapReduce programs into ALP. Similar extensions are set to enlarge the scope of ALP applicability and are the subject of ongoing research. It is important that such extensions remain minimal as well as faithful, that is, the core set of ALP primitives should remain as few as possible, while ALP containers, structures, and primitives must have clear corresponding concepts in mathematics.

4.3 Expand

However well programmers may prove to be in handling algebraic concepts explicitly, and however broad the scope of applicability of the ALP paradigm may prove to become, the humble mindset dictates that programmers should always be allowed to use the tools that *they* consider most intuitively suitable for different programming tasks. Nevertheless, unchecked growth in software stacks supporting different programming paradigms, many architectures, and many heterogeneous configurations, does not scale. This implies that the number of programming tools with distinct software stacks should be minimized.

Breaking the paradox of these seemingly conflicting demands, this paper proposes that humble programming models can be automatically expanded into other, more *fundamental* humble programming models. With such a mindset, many humble programming models could exist, though only very few should be fundamental. The many humble models should automatically translate to a fundamental one, and only fundamental programming models should be backed by an automatically optimizing, parallelizing, and ideally architecture-portable software stack.

This paper prototypes this vision by automatically translating the successful, scalable, and humble vertex-centric programming model Pregel [MAB+10] into the ALP paradigm. It shows how automatic translation enables multiple humble programming paradigms that share the same software stack, demonstrates both the performance and scalability of this approach, and consequently proposes ALP as a fundamental humble programming model. With this solution, humble programmers can rely on multiple programming interfaces and select the most suitable ones for each job, while hero programmers can focus their efforts on a limited number of fundamental programming models and software stacks.

5 ALP/Pregel

This section first describes the programming interface for a vertex-centric programming model on top of ALP/GraphBLAS. It is a pure vertex-centric interface that does not expose ALP concepts, except for a commutative monoid over which incoming messages are to be reduced. Using this interface, the SCC algorithm is revisited and precisely formulated, and an ALP/Pregel program is introduced for web page ranking based on the canonical PageRank algorithm [PBMW99].

5.1 Interface

The C++ interface supports two termination mechanisms: vote-to-halt and inactivation of vertex programs. For the former, all vertex programs vote on whether to halt the program, which is terminated only if all active vertices vote to halt it. For the latter, vertex programs can set themselves inactive, after which point they shall no longer participate in subsequent rounds. If all vertex programs are inactive, then the overall program terminates.

Algorithm 1 shows the SCC algorithm in our vertex-centric API, and Algorithm 2 shows a simplified PageRank algorithm. Both algorithms are located in the `grb::algorithms::pregel` namespace and provide a concise way of introducing the ALP/Pregel interface.

For both examples, the vertex-centric program corresponds to the `program` static member function defined. Each program 1) operates on given vertex data of a program-defined type, 2) expects an incoming message of a potentially different type, 3) generates an outgoing message of a potentially third different type, 4) has read access to program-specific data and other parameters, and 5) has access to a predefined PregelState type instance providing read access to Pregel metadata and write access to Pregel termination controls.

5.2 SCC Algorithm

Briefly summarizing the earlier SCC example for undirected graphs, every vertex is initially assigned a unique identifier (ID) integer. Each vertex then broadcasts its ID, overwriting its local ID by the maximum ID received. If the current ID is already the maximum and no update is performed, the program votes to halt the execution. One may intuitively find that, in line with being a humble programming model, this algorithm indeed converges to a correct solution where

Algorithm 1 Vertex-centric SCC algorithm

```
template< typename VertexIDType >
struct ConnectedComponents {

        struct Data {};

        static void program(
                VertexIDType &current_max_ID,
                const VertexIDType &incoming_message,
                VertexIDType &outgoing_message,
                const Data &parameters,
                grb::interfaces::PregelState &pregel
        ) {
                if( pregel.round > 0 ) {
                        if( pregel.indegree == 0 ) {
                                pregel.voteToHalt = true;
                        } else if( current_max_ID < incoming_message ) {
                                current_max_ID = incoming_message;
                        } else {
                                pregel.voteToHalt = true;
                        }
                }
                if( pregel.outdegree > 0 ) {
                        outgoing_message = current_max_ID;
                } else {
                        pregel.voteToHalt = true;
                }
        }

};
```

every component is assigned a unique ID that corresponds to the maximum of values initially assigned to all vertices in the component.

The ID type may be any integer, such as **unsigned int** or **size_t**, and incoming and outgoing messages are of the same type. The algorithm does not require any algorithm-specific parameters (hence the empty Data class on line 4 of Algorithm 1). It exemplifies the use of the in-degree (line 2 of the program body) and out-degree (line 10 of the program body) metadata that our Pregel API provides, and makes use of the vote-to-halt mechanism. The program terminates within d steps, where d is the maximum diameter of all components in G.

Execution of the strongly connected algorithm requires the Pregel interface to be initialized over a specific graph. Conversely, in ALP/GraphBLAS, a graph file is typically opened using a parser and then ingested into a `grb::Matrix` instance, for example, by

```
grb::Matrix< void > A( parser.m(), parser.n(), parser.
nz() );
grb::buildMatrixUnique( A, parser.begin(), parser.end()
);
```

The same graph file should now be ingested into a Pregel instance:

```
grb::interfaces::Pregel< void > pregel(
        parser.n(), parser.m(),
        parser.begin(), parser.end()
);
```

The **void** template parameter to the Pregel interface indicates that edge weights must be ignored. This is because Pregel algorithms determine the message patterns based on the edge structures. Nevertheless, the template argument remains for future extensions that may be considered edge-centric, or vertex- and edge-centric, programming paradigms.

Once the Pregel instance is instantiated, it may execute any Pregel algorithm on the underlying graph. The execution employs the execute member function of the Pregel

Algorithm 2 Vertex-centric PageRank algorithm

```
template< typename IOType >
struct PageRank {

        struct Data {
                IOType alpha = 0.15;
                IOType tolerance = 0.00001;
        };

        static void program(
                IOType &current_score,
                const IOType &incoming_message,
                IOType &outgoing_message,
                const Data &parameters,
                grb::interfaces::PregelState &pregel
        ) {
                // initialise
                if( pregel.round == 0 ) {
                        current_score = static_cast< IOType >(1) /
                                static_cast< IOType >(pregel.num_vertices);
                }

                // compute
                if( pregel.round > 0 ) {
                        const IOType old_score = current_score;
                        current_score = parameters.alpha +
                                ( static_cast< IOType >(1) - parameters.alpha ) * incoming_message;
                        if( fabs(current_score-old_score) < parameters.tolerance ) {
                                pregel.active = false;
                        }
                }

                // broadcast
                if( pregel.outdegree > 0 ) {
                        outgoing_message = current_score / static_cast< IOType >(pregel.outdegree);
                }
        }
};
```

instance. For example, in the case of starting Algorithm 1,

```
const size_t n = pregel.num_vertices();
const size_t max_steps = n;
size_t steps_taken;
grb::Vector< size_t > component_IDs( n ), in_msgs( n ),
out_msgs( n );
grb::RC error_code = pregel.template execute<
        grb::operators::max< VertexIDType >,
        grb::identities::negative_infinity
> (
      &(grb::algorithms::pregel::ConnectedComponents::
program),
      component_IDs,
        grb::algorithms::pregel::ConnectedComponents::
Data(),
      in_msgs, out_msgs,
      steps_taken, max_steps
);
```

the execute function is a templated member function whose template arguments define the commutative monoid under which incoming messages are aggregated. Its domains must match that of the outgoing and incoming messages. For SCC, all domains should be of the same integer type. For a specific vertex, using a monoid structure rather than an arbitrary message combiner operator helps ensure the following:

- In cases when no incoming messages are received (e.g., because a vertex in-degree is zero or all vertices with edges incident to this vertex have become inactive), the monoid identity can be substituted as a received message, thus ensuring consistent behavior while monoid associativity ensures scalable reduction.
- In cases when multiple messages are received in an order that potentially differs across rounds and executions, the commutativity of the combiner monoid ensures consistent behavior.

In the SCC example, we selected the maximum component ID. As such, Algorithm 1 demonstrates how the max-monoid for message aggregation is passed to the executor. The non-template arguments include, in order: 1) the vertex-centric program, 2) the vertex states as a dense vector, 3) the program parameters, 4) buffers for the incoming and outgoing messages, and 5) an output field that records the number of rounds the program took before termination. An optional argument, max_steps, limits this number of rounds. If max_steps is not provided, the program will run until a termination condition arises.

The error_code `grb::SUCCESS` will be returned if the algorithm terminates correctly, and `grb::FAILED` will be returned if the algorithm did not reach a termination condition after the maximum number of rounds was completed. `grb::MISMATCH` will be returned if the message buffers do not match the size of the number of vertices in the underlying graph, and `grb::ILLEGAL` will be returned if any of the passed vectors do not have full capacity.

5.3 Vertex-centric PageRank Algorithm

In the second example, the vertex-centric PageRank in Algorithm 2 is simplified to the point where it no longer corresponds to the canonical algorithm [PBMW99]. Despite this, it is a common approximation that appears in vertex-centric and Spark-based literature, for example, Gonzalez et al. [GXD+14]. The algorithm has two canonical algorithm parameters: α and tol. The former, α, can be viewed as the regularization parameter, or more intuitively, the probability that someone viewing a web page visits another web page at random rather than via a link on the current page. The latter, tol, signifies a desired local tolerance. Once it is met, the current vertex will be set to inactive.

The local state of every vertex is its PageRank score, represented here by the value of IOType (e.g., **double**). In each round, the vertex-centric program distributes the local score equally to all neighbors of the current node. Specifically, the local score s is divided across all neighboring vertices as an outgoing message with value s/d, where d is the out-degree. Incoming messages are aggregated via the standard additive monoid, resulting in an incoming score of s'. Finally, the vertex-centric program determines the new local score as $\alpha + (1 - \alpha)s'$. If the difference with the previous local score is less than tol, the program considers the local vertex converged and removes it from further rounds of computation. In this case, the local vertex's active neighboring nodes assume that the now-inactive vertex keeps sending the aggregator monoid identity as its outgoing message.

In this example, the out-degree of each vertex is used in computing an outgoing message. This example demonstrates the use of the `PregelState::round` field — used to keep track of the current round of computation — to initialize the vertex weights during the first round (lines 1–5 of the program body). Furthermore, this example demonstrates that in initialization, global information on the total number of vertices in the program (`PregelState::num_vertices`) can be employed within vertex-centric programs by normalizing the initial PageRank score[3]. The inner `if`-statement of the compute block shows how a vertex removes itself from computation. And, in this example, no novel concepts are introduced as to the

[3] This normalization is didactic – it is not necessary for this PageRank-like algorithm to converge.

mechanisms used to broadcast outgoing messages (lines 17–20 of the program body).

As shown in the preceding example, the PageRank program is executed on a Pregel instance. However, the vertex weights and messages are of type **double** instead of **size_t**, and a regular additive monoid is used instead of a max monoid. As per convention, ALP/Pregel algorithms also provide a static member `function::execute` that constructs the necessary communication buffers and monoid definition. With a Pregel instance constructed over some input graph as before, the following executes the ALP/Pregel PageRank algorithm using the default algorithm parameters and an unlimited number of rounds:

```
grb::Vector< double > pr_scores( n );
grb::set( pr_scores, 0 );
grb::algorithms::pregel::PageRank< double >::execute(
     pregel, pr_scores, rounds_taken
);
```

Theoretically, termination of this program is not guaranteed, especially when rounds are limited to a small number or when a low α is chosen [LM11]. Section 6 takes care to report only experiments where program executions terminate successfully.

As noted earlier, this PageRank algorithm does not correspond to the canonical version by Page et al. in that it lacks contributions from the dangling nodes[4]. A common extension to the original Pregel framework [MAB+10] adds global aggregation mechanisms, which would allow for correct implementation of the PageRank algorithm. The addition of these mechanisms corrects the PageRank updates with contributions from the dangling nodes and enables global convergence detection. Such mechanisms are provided, for example, by the Giraph [Ave11] vertex-centric framework. ALP/Pregel, however, does not provide such mechanisms, as the author believes they will limit the potential of automatic overlap of message exchanges with the execution of rounds. This is discussed in more detail in Section 7.2.

ALP/Pregel supports permanently removing vertices from execution by setting them inactive. Although this is a known optimization for improving convergence for the PageRank algorithm [LM11], it is not supported by all vertex-centric frameworks. While this mechanism may accelerate convergence for the PageRank algorithm, ALP/Pregel also exploits inactive vertices to accelerate per-round computations, as described in Section 5.4. The examples thus far have introduced all fields available in `grb::interfaces::PregelState`, except for num_edges.

[4] Nodes with out-degree zero.

Table 1 summarizes all available fields.

Table 1 All fields available to a vertex-centric program during computation rounds. The fields are sorted from writable ones that control program termination, to read-only global constants and variables, and finally to read-only constant numbers regarding local vertex properties.

Field name	read or write	Description
active	write	Set to false to become inactive in subsequent rounds
voteToHalt	write	Set to true to vote for halting the program
round	read	The current computation round
num_vertices	read	The total number of vertices in the current graph
num_edges	read	The total number of edges in the current graph
indegree	read	The in-degree of the current vertex
outdegree	read	The out-degree of the current vertex
vertexID	read	The unique ID of the current vertex

5.4 Implementation

The implementation of the vertex-centric ALP/Pregel interface translates the program to a while-loop around standard ALP/GraphBLAS primitives at compilation time. Each execution of the body of the while loop corresponds to the execution of one round of computation as well as the subsequent termination detection and message exchange. Prior to the first round, all vertices are added to the *active list*, and the outgoing message buffer is reset to the monoid identity. During each round, the following operations take place for vertices that are in the active list:

1 reset the outgoing message buffers using the aggregation monoid identity;

2 call the user-defined vertex-centric program, passing in the current vertex state the program operates on, the buffers for incoming and outgoing messages, the algorithm-specific global data (e.g., α and tol for the vertex-centric PageRank), and the PregelState instance;

3 determine whether all active vertices have voted to halt;

4 remove the vertices that have set themselves inactive from the active list;

5 determine whether all vertices have become inactive;

6 increment the round counter;

7 reset the incoming message buffers using the aggregation monoid identity; and

8 exchange messages and aggregate incoming messages via `grb::vxm`.

Steps 5–8 use the updated active list from step 4.

Data structures. The active list, incoming messages, outgoing message, vertex states, in-degrees, out-degrees, and vertex IDs are all maintained as `grb::Vectors` with non-zeroes of the following types: **bool** for the active list, user-defined for the message and state vectors, and **size_t** for the degree and ID vectors. All vector sizes equal the number of vertices in the graph, n. On initialization of a `grb::interfaces::Pregel` instance, the active list is initialized to **true** for all vertices, and the in-degrees and out-degrees are computed using SpMV multiplication of the graph adjacency matrix with a vector of ones. The vertex IDs are computed via

```
grb::set< grb::descriptors::use_index >(
    vertexIDs, 0 );
```

The use_index descriptor writes the index *i* to each element of vertexIDs, instead of the given scalar value 0.

The values of the outgoing (and incoming) message vector are reset each round via

```
grb::set< grb::descriptors::structural >(
    outgoingMessages, activeList, id
);
```

Here, `id` is the identity of the aggregation monoid, which is of the same type as that of outgoing messages. Note that the preceding code resets only the `outgoingMessages` of active vertices and throws away entries corresponding to inactive vertices.

Calling the user program. The user program executes through a call to `grb::eWiseLambda`. The lambda function passed into the eWiseLambda captures a reference to the active list, incoming and outgoing messages, state, in-degrees and out-degrees, and vertex IDs. It then calls the user-defined program. The eWiseLambda only executes for the vertices that remain active in every round by passing activeList as the leading mask of the element-wise lambda primitive.

Updating the active list. For efficiency, the active list should be maintained as a sparse vector where the number of non-zeroes equals the number of active vertices at the start of each round. All masked operations can then take the structural descriptor, which prevents touching the actual Boolean values of each entry in the active list. However, the user must be allowed to set an active node to **false**, which does require that actual Boolean values be present in the active list. Therefore, the update of active vertices takes place directly after calling the user program, and is implemented using `grb::set (buffer, activeList, true)` *without* passing in a structural descriptor. This operation is the only step primitive in the ALP/Pregel implementation without a structural descriptor. The use of the masked set operation ensures that there are fewer non-zeroes in the buffer. Finally, the buffer is swapped with the active list to be used as the new mask for subsequent steps of the algorithm.

An extra buffer is maintained to support this update step. According to the performance semantics defined by ALP/GraphBLAS, the `grb::set` on a non-empty container implies a cost proportional to the number of non-zeroes before the update, plus a cost proportional to the number of non-zeroes after the update [YDNNS20]. As such, while the buffer takes up $\Theta(n)$ memory, the overhead of this update step is bounded as $\Theta(k)$, with k being the number of active vertices before the update.

Termination detection. Termination detection using the vote-to-halt mechanism takes place via a reduction of the voteToHalt vector into a Boolean scalar using the logical AND monoid, whereas termination detection using the inactive mechanism takes place by counting the number of non-zeroes in the updated active list.

Message exchange. Describing how to perform message exchange and aggregation using vector–matrix multiplication may be unclear if the semiring under which this proceeds is not described. The given aggregation monoid functions as the additive monoid of such a semiring, from whence the requirement that the aggregation monoid needs to be a commutative monoid. The semiring multiplicative operator is

```
grb::operators::left_assign_if<
        IncomingMessageType, bool, IncomingMessageType
>
```

with the Boolean **true** as its identity element. Denoting the application of this multiplicative monoid operation as $x \otimes y = \mathit{left\_assign\_if}(x, y)$, with x from a user-defined domain D and $y \in \{\mathit{false}, \mathit{true}\}$, then $\mathit{left\_assign\_if} : D \times \{\mathit{false}, \mathit{true}\} \to D$ is defined as follows:

$$x \otimes y = \mathit{left\_assign\_if}(x, y) = \begin{cases} x, \text{ if } y \text{ equals } \mathit{true} \\ \mathbf{0}, \text{ otherwise.} \end{cases}$$

In the latter case, **0** corresponds to the additive monoid identity.

The resulting semiring is, in fact, an improper one — because the aggregator identity may be user-defined, it may not annihilate under the multiplicative operator. However, if the additive monoid is the logical OR with **false** as the additive identity, the resulting semiring with *left_assign_if* is, in fact, proper since it reduces to the standard Boolean semiring.

The way we use semiring guarantees that the multiplicative operator is only ever called using the multiplicative identity as the right-hand side input argument, thereby ensuring that improper cases for non-logical-OR aggregators are not triggered. This must be proceeded carefully as follows: The `grb::vxm` operation $in = outG$ matches non-zeroes $g_{ij} \in G$ to corresponding elements out_i from the outgoing message buffer, and first applies the multiplicative operator to form a temporary $tmp = out_i \otimes G_{ij}$. Because G is a **void** matrix, g_{ij} resolves to the semiring identity, **true**, so that

$$tmp = out_i \otimes true = assign\_left\_if(out_i, true) = out_i.$$

This temporary value is then aggregated into in_j using the user-defined aggregator monoid. The ALP/Pregel implementation thus ensures that explicit multiplication with zero (which may not annihilate) never occurs.

5.5 Summary and Costing

The implementation of vertex-centric programming in ALP/GraphBLAS makes ample use of its main features:

- composability of monoids and semirings provides the flexibility to integrate user-defined aggregation into the richer algebraic semiring structure;
- `grb::eWiseLambda` allows the execution of any user-defined operation on the vertex data;
- sparsity, masks, and algebraic structures are exploited to achieve high performance automatically.

The implementation requires one ALP/GraphBLAS vector container for each of the entries in Table 1, as well as one buffer vector for the active list. Through these containers, ALP/Pregel uses $\Theta(n)$ memory for executing any Pregel program. Execution neither allocates nor frees any memory. Computing termination conditions costs $\Theta(k)$ work, with k being the number of active vertices in a given round. In the worst-case scenario, capturing vector elements as arguments to user-defined vertex-centric programs has $\mathcal{O}(1)$ overhead, again translating to $\mathcal{O}(k)$ costs per round. The cost of executing a vector program is determined by the user, and is linear with k. Specifically, for a vertex-centric program that locally takes $\Theta(1)$ time, executing all active programs costs $\Theta(k)$ work.

Again in the worst-case scenario, data movement complexities in a shared-memory setting correspond to $\Theta(k)$ with regard to accessing internal vector states. The persistent state of a single vertex and how much of it is touched during each round is user-defined. However, if we assume $\Theta(1)$ memory movement by a vertex-centric program during any round, the total memory movement is also $\Theta(k)$ per round. Assume that the sizes of incoming and outgoing messages are $\Theta(1)$, the message exchange costs

$$\mathcal{O}\left(k + \sum_{i \in activeList} d_i\right) \tag{1}$$

data movement per round, where d_i is the in-degree of the i-th vertex. Although perhaps counterintuitive, exchanging messages also implies work. For ALP/GraphBLAS, the work bound equals that of Eqn. (1), except that the bound is big-Theta instead of big-Oh.

For shared-memory parallelization, the above work bounds may be divided by T, where T is the number of threads. Parallelization also adds a factor $\mathcal{O}(T)$ to all storage, work, and data movement bounds. Consequently, the work bound corresponds to the per-thread work, while data movement bound considers the data volume moved across the whole system.

Likewise, for distributed-memory parallelization, work can be divided by P, which is the number of distributed-memory processes, while adding a factor $\mathcal{O}(P)$ to all storage, work, and data movement bounds. Additionally, the active list update and the vote-to-halt termination check each amount to a collective reduction across all nodes. This induces $\mathcal{O}(P)$ inter-process data movement and work, and as many as $\mathcal{O}(\log P)$ synchronization steps, where P is the number of distributed processes. The message exchange adds $\mathcal{O}(k)$ inter-process data movement, assuming an $\Theta(1)$ storage for the outgoing messages. Tables 2 and 3 summarize all costs.

These per-round costings are a direct application of the ALP/GraphBLAS performance semantics [YDNNS20]. They confirm that, for an increasing number of inactive vertices, bounds for work and both intra- and inter-process data movement improve.

Table 2 Costs of executing a single round of an ALP/Pregel program on a shared-memory machine. This assumes there are n vertices, of which k are active, m edges, $\Theta(1)$ execution time of a vertex-centric program, and $\Theta(1)$ storage for each of a single vertex state, incoming message, and outgoing message. The work corresponds to that of a single thread, and the data movement corresponds to that across the entire machine. The sum $\sum_i d_i$ is as in Eqn. (1), while T indicates the number of threads within a shared-memory machine. Sequential costs correspond to $T = 1$.

	Storage	Work	Data movement
Round computation	$\Theta(n+T-1)$	$\Theta(k/T+T-1)$	$\Theta(k+T-1)$
Active list update	$\Theta(n+T-1)$	$\Theta(k/T+T-1)$	$\Theta(k+T-1)$
Termination check	$\Theta(T)$	$\mathcal{O}(k/T+T-1)$	$\mathcal{O}(k+T-1)$
Message exchange	$\Theta(n+m+T-1)$	$\Theta((k+\sum_i d_i)/T+T-1)$	$\mathcal{O}(k+[\sum_i d_i]+T-1)$
Total	$\Theta(n+m+T-1)$	$\Theta((k+\sum_i d_i)/T+T-1)$	$\mathcal{O}(k+[\sum_i d_i]+T-1)$

Table 3 Additional costs of executing a single round of an ALP/Pregel program on a distributed-memory architecture. The costs are in addition to those listed in Table 2, only that T should be replaced with the number of threads available at each process multiplied by the number of distributed-memory processes P. Additionally, all its data movement costings should be divided by P, and each of storage, work, and data movement adds a factor $\mathcal{O}(P)$.

	Work	Data movement	Synchronizations
Round computation	0	0	0
Active list update	$\mathcal{O}(P)$	$\mathcal{O}(P)$	$\mathcal{O}(\log P)$
Termination check	$\mathcal{O}(P)$	$\mathcal{O}(P)$	$\mathcal{O}(\log P)$
Message exchange	$\Theta(k)$	$\Theta(k)$	1
Total	$\Theta(k)+\mathcal{O}(P)$	$\Theta(k)+\mathcal{O}(P)$	$\mathcal{O}(\log P)$

6 Experiments

Experiments are conducted for two purposes: 1) to show that humble ALP/Pregel programs can achieve the scalability predicted in the previous section, and 2) to show that ALP/Pregel is competitive against state-of-the-art frameworks. For the first purpose, the SCC algorithm is investigated, comparing the sequential and parallel results. Then, using two variants of the PageRank algorithm, the behavior of ALP/Pregel under an increasing number of inactive vertices is investigated. Finally, a scalability experiment using PageRank algorithms demonstrates that shared-memory auto-parallelization of ALP/Pregel behaves as expected also when the number of inactive vertices increases.

For the second purpose of comparing ALP/Pregel with state-of-the-art frameworks, this section first summarizes the ALP/GraphBLAS performance from the recent work by Mastoras et al. [MAY23], who provide an in-depth comparison of ALP/GraphBLAS against the GNU Scientific Library (GSL), Eigen [GJ+10], and SuiteSparse:GraphBLAS [Dav19]. They show that the ALP/GraphBLAS PageRank implementation sketched in Algorithm 3 is 0.96–9.82 times faster than a PageRank implemented in SuiteSparse:GraphBLAS, and 0.64–14 times faster than one written using Eigen on shared-memory parallel architectures. Both SuiteSparse and Eigen auto-parallelize over shared-memory architectures, while GSL does not. Eigen additionally performs loop fusion, which leads to speedups over the blocking ALP/GraphBLAS for small matrices. The nonblocking ALP/GraphBLAS implementation that Mastoras et al. contribute also achieves loop fusion and achieves speedups beyond Eigen for both smaller and larger matrices, but will not be considered as part of the experiments in this section. Mastoras et al. obtain similar results for two other sparse matrix and graph algorithms.

This paper focuses on comparing the ALP/Pregel PageRank-like algorithm with the canonical ALP/GraphBLAS variant, which is taken as the state-of-the-art baseline. This paper does not compare ALP/Pregel against the existing vertex-centric frameworks such as Giraph [Ave11], since such frameworks typically exhibit orders-of-magnitude performance losses versus the optimized code [SY19] because they rely on file-based fault tolerance. Such comparisons are out of our scope because what we look for are humble programming models that achieve hero-level performance.

6.1 Methodology

Experiments are run on a dual-socket Intel x86 machine, consisting of two Intel Xeon 6238T processors, each having 22 cores, a 32 kB private L1 cache per core, a 1 MB private L2 cache per core, and a 30.25 MB shared L3 cache. The cores, clocked at 1.9 GHz, have hyperthreading enabled and turbo boost disabled. Each processor has six memory channels clocked at 2,933 MT/s, producing 262.2 GB/s of theoretical throughput across the total machine. The combined computational throughput of all AVX-512 enabled cores is 2,675 Gflop/s. The single-core memory throughput ranges from 9.95 (vector-to-scalar reduction) to 18.4 (triad) Gbyte/s. These figures are relevant because ALP/Pregel computations are typically memory-bound, achieving far lower speedup than the total number of cores would indicate. Indeed, for this architecture, a bandwidth-bound application is expected to achieve 14.3–26.4 times of speedup.

Our experiments are implemented on v0.6 of ALP/GraphBLAS, which is available on GitHub and Gitee [Alg21a, Alg21b]. Its sequential reference and shared-memory parallel reference_omp backends are used both to compile ALP/Pregel programs and to provide a baseline PageRank implementation. All software is compiled using GCC 9.3.1 with the **-O3 -mtune=native-march=native -DNDEBUG -funroll-loops** compiler flags. All compilations and experiments are executed on Linux kernel version 5.8.18. Experiments using OpenMP define **OMP_PROC_BIND=true**.

Experiments on small datasets such as gyro_m are repeated multiple times to ensure that one timing takes at least 100 milliseconds. Then, experiments are further repeated at least 10 times in order to compute the sample standard deviation across all timings. All reported timings have a sample standard deviation of less than 3% of the measured average time. Timings with sample standard deviation being 1% higher than the average time are printed in *italics*, and the tables only show three significant digits of each of the obtained averages. As such, timings printed in non-italics are accurate to a significant degree, while for timings printed in italics the least significant digit is likely to be inaccurate.

In this paper, the presented methodology is only not applied to three experiments benchmarking the sequential ALP/Pregel SCC due to a very long run time.

6.2 Datasets

Our experiments require input graphs on which to run the vertex-centric algorithms. Typical datasets that vertex-centric frameworks operate on include knowledge graphs and graph representations from the Web and other types of networks. To also compare against structures that differ significantly from typical graphs so that we can gauge the effectiveness when faced with problems originating from other domains, we include both graphs and sparse matrices from various scientific computing domains in our dataset, as shown in Table 4.

Table 4 Graphs used in the experiments. All except 11 and 12 are undirected.

Dataset	Name	#Vertices	#Edges	Edges/Vertex	Structured
1	gyro_m	17 361	340 431	19.6	no
2	vanbody	47 072	2 336 898	49.6	yes
3	G2_circuit	150 102	726 674	4.84	mixed
4	bundle_adj	513 351	20 208 051	39.4	adverse
5	apache2	715 176	4 817 870	6.74	yes
6	ecology2	999 999	4 995 991	5.00	yes
7	Emilia_923	923 136	41 005 206	44.4	yes
8	Serena	1 391 349	64 531 701	46.4	yes
9	G3_circuit	1 585 478	7 660 826	4.83	mixed
10	Queen_4147	4 147 110	329 499 284	79.5	yes
11	wikipedia-20070206	3 566 907	45 030 389	12.6	no
12	uk-2002	18 520 486	298 113 762	16.1	no
13	road_usa	23 947 347	57 708 624	2.41	no

All datasets are from the SuiteSparse MatrixMarket collection [DH11], and are ordered by the number of vertices they contain. In the architecture described earlier, 128k 64-bit words (**double**s or **size_t**s) fit into private L2 caches. As such, we expect significant data reuse by algorithms operating on datasets 1 and 2. Approximately 4M words fit into shared L3 caches. Given that all algorithms require multiple vectors to operate, significant reuse of data in L3 caches should occur for datasets 3–10. Datasets above number 10 will always induce out-of-cache behavior because the corresponding vectors do not fit in L3 caches. As we will describe later in this section, performance differences between some PageRank implementations results from the difference in the number of edges per vertex (also reported in Table 4).

Apart from differences in the number of vertices and edges, we augment graphs that have no discernible structure when plotting its corresponding adjacency matrix with structured matrices. Discernible structures in such matrices include having a fixed number of diagonals in the adjacency matrix, and with only non-zeroes within a fixed bandwidth around the main diagonal. Structured matrices and graphs with structured corresponding adjacency matrices induce favorable data access patterns that modern hardware prefetchers implicitly exploit, which results in very different behavior during computations on such graphs. These effects are well-known in high-performance computing research that deals with sparse matrices. Some datasets, such as G3_Circuit, have both structured and unstructured components in the adjacency matrix, whereas bundle_adj has an adversarial structure that has a major impact on the performance of linear algebraic libraries and programming frameworks [MAY23]. All graphs represented by the adjacency matrices are undirected, except wikipedia-20070206 and uk-2002.

6.3 SCC Algorithm

We first consider the scalability of ALP/Pregel using the SCC for undirected graphs in Algorithm 1. For directed graphs this algorithm still terminates, and returns connected components where for each vertex v in a given component, there exists at least one other vertex u in that same component with a path to u to v. Given Algorithm 1 over directed inputs retains a meaningful interpretation, our experiments use the full dataset in Table 4.

Table 5 compares the wall-clock time of executing the related ALP/Pregel SCC algorithm between two modes: using the sequential ALP/GraphBLAS backend and using the shared-memory parallel OpenMP-enabled backend. We emphasize that parallelization is fully automatic and the code presented in Algorithm 1 does not need to be changed.

Table 5 The runtime in milliseconds for executing the ALP/Pregel SCC, Algorithm 1, compiled using the sequential and shared-memory parallel ALP/GraphBLAS backends. The numbers of iterations are listed in the parentheses following each timing, and the last column reports the speedup of parallel execution over sequential execution.

Dataset	Sequential	Shmem. parallel	Speedup
1	39.3 (47)	38.6 (47)	1.03×
2	183 (53)	56.2 (53)	3.26×
3	755 (162)	255 (162)	2.96×
4	4 910 (140)	568 (140)	8.64×
5	11 100 (447)	1 970 (447)	5.63×
6	52 300 (2000)	11 400 (2000)	4.59×
7	*6 930* (111)	774 (111)	8.95×
8	*6 090* (59)	618 (59)	9.84×
9	27 400 (523)	4 500 (523)	6.09×
10	*76 200* (178)	5 620 (178)	13.6×
11	189 000 (461)	10 800 (461)	17.5×
12	*131 000* (116)	11 900 (116)	11.0×
13	6 080 000 (5681)	668 000 (5681)	9.10×

The number of rounds required by the algorithm is shown in the parentheses. In addition, the table also reports speedup obtained by shared-memory parallelization.

The SCC algorithm employs only the vote-to-halt termination mechanism so that each round runs with all vertices being active. Moreover, parallelization should not affect the number of rounds computation requires. This experiment therefore measures only the scalability of the shared-memory parallel backend using the SCC algorithm.

Parallelization involves unavoidable overheads of at least $\Omega(\log T)$. In ALP/GraphBLAS, it is bounded by $\mathcal{O}(T)$ [YDNNS20]. Therefore, it is certain that for smaller datasets on a full system with 88 hyperthreads and two NUMA domains, speedups are expected to be far below 26.4x, which is achieved by the triad benchmarks[5]. Since the vote-to-halt termination mechanism depends on the vector-to-scalar reduction, a significant portion of the vertex-centric paradigm will be bound by the 14.3x of speedup for the reduction benchmark[5]. However, as the size of datasets increases, speedups are expected to reach the 14.3–26.4 range.

Table 5 reports the same number of rounds for both the sequential and shared-memory parallel ALP/Pregel SCC algorithms. The maximum speedup, 17.5x for dataset 11, falls within the upper range, indicating that parallelization of the non-reduction parts of the ALP/Pregel program achieves speedups closer to the best-case triad benchmark. This indicates that the ALP/Pregel implementation scales on shared-memory architectures for the particular case when all vertices remain active throughout the computation.

6.4 Sequential PageRank

By contrast, the PageRank in Algorithm 2 does make use of the inactive vertices. In this case, as the computation proceeds, fewer and fewer vertices will partake in the computation. Termination of this program does not guarantee convergence in the classical 2–norm, nor in the inf-norm, even if dangling nodes were accounted for properly.

To compare the impact of the vertex inactivation mechanism on the speed of computation, we introduce a global variant of the ALP/Pregel PageRank by modifying Algorithm 2. Specifically, instead of using the inactivation mechanism, the algorithm is modified to use vote-to-halt on local convergence detection. As such, we subsequently refer to the original Algorithm 2 as the local variant, because it does not consider a global inf-norm computation but a greedy local version of it.

Both the global and local ALP/Pregel PageRank algorithms are further compared against the standard PageRank implementation using and bundled with ALP/GraphBLAS. This algorithm does account for dangling nodes and implements convergence detection in the standard 2–norm [YDNNS20]. Algorithm 3 provides a simplified listing of that algorithm for the sake of completeness[6]. Because of the algorithmic differences relating to dangling nodes, the standard equivalence of norms does not apply, and we therefore cannot directly compare errors between the ALP/Pregel implementations and the ALP/GraphBLAS implementation (the algorithms are different and converge to different values). As described in Section 5.3, the vertex-centric PageRank is commonly applied, and its global variant is most often applied. Table 6 therefore compares the local vertex-centric variant against its global variant in terms of performance and speed of convergence. It also compares both Pregel variants and the canonical PageRank approaches to web page ranking in terms of performance.

In terms of the number of iterations, among the three different algorithms we compare, it is not the case that the vertex-centric algorithms always incur fewer iterations than the canonical PageRank, nor that the local variant of the ALP/Pregel PageRank always incurs fewer iterations than its global variant. Indeed, as Table 6 shows, any algorithm can incur the fewest number of iterations, depending on the input graphs. The vertex-centric variants do require significantly more iterations for the large undirected graphs, IDs 11 and 12, to converge, but this may well be due to the common presence of dangling nodes in those graphs.

In terms of the execution speed, the local variant is faster than the global variant. This is because, as the rounds progress, increasingly more vertices become inactive. This is confirmed by the time per iteration reported in Table 6. As such, we may conclude that our ALP/Pregel implementation becomes faster as the number of active nodes decreases. One disadvantage of using the element-wise lambda noted by Yzelman et al. is that the compiler is no longer able to apply vectorization when possible [YDNNS20]. Thus, the ALP/GraphBLAS PageRank may be faster than the global ALP/Pregel variant even though the former performs more operations[7]. This benefit is expected to decrease as the number of edges per vertex increases.

[5] See Section 6.1.

[6] See the ALP/GraphBLAS repository [Alg21a] for the full algorithm.

[7] i.e., dangling node corrections and 2–norm computations.

Table 6 Results, in milliseconds, of executing sequential PageRank algorithms, comparing the ALP/Pregel variants using global and local convergence against the baseline ALP/GraphBLAS PageRank implementation. The numbers of iterations implemented before convergence are listed in the parentheses. The last three columns report the time per iteration, with the fastest marked in **bold**.

ALP/Pregel				ms. per iteration		
Dataset	**Global**	**Local**	**Baseline**	**Global**	**Local**	**Baseline**
1	34.8 (40)	24.7 (39)	31.4 (52)	0.870	0.633	**0.604**
2	192 (43)	118 (41)	197 (52)	4.47	**2.88**	3.79
3	175 (38)	78.8 (36)	90.0 (48)	4.61	2.19	**1.88**
4	3 070 (66)	2 070 (51)	2 330 (60)	46.5	40.6	**38.8**
5	*707* (31)	456 (33)	*434* (43)	22.8	13.8	**10.1**
6	976 (31)	63.5 (34)	*375* (30)	31.5	**1.87**	12.5
7	2 840 (36)	1 180 (36)	*2 960* (45)	78.9	**32.8**	65.8
8	5 090 (40)	*2 500* (33)	4 750 (44)	127	**75.8**	108
9	1 960 (38)	987 (36)	*1 100* (48)	51.6	27.4	**22.9**
10	20 800 (35)	*2 780* (35)	25 000 (46)	594	**79.4**	543
11	40 500 (103)	11 400 (96)	18 100 (55)	393	**119**	329
12	153 000 (115)	*46 100* (104)	*72 100* (73)	1330	**443**	988
13	*87 600* (78)	58 800 (72)	62 200 (78)	1120	817	**797**

The time per iteration reported in the preceding table shows that the ALP/GraphBLAS outperforms the global variant, and even the local vertex-centric variant in 6 out of 13 cases. For the larger graphs, the latter is explained by the observation that low edge count per vertex favors vectorization (datasets 6, 9, and 13). Additionally, a performance gain for the pure ALP/GraphBLAS algorithm on small graphs is observed. This gain increases on the shared-memory parallel ALP/GraphBLAS variant, as presented and analyzed in more detail in the next section.

In summary, in terms of the time per iteration, the local ALP/Pregel PageRank is up to 16.8x faster than its global variant, and up to 6.84x faster than a state-of-the-art canonical PageRank algorithm. In terms of the end-to-end runtime, the local variant is up to 7.48x faster than the global one, and up to 8.99x faster than the state-of-the-art baseline — the biggest slowdown is limited at 0.95x. The local ALP/Pregel algorithm is fastest overall in 12 out of 13 cases.

6.5 Shared-memory Parallel PageRank

Similar to the SCC algorithm, in the case of parallelization of PageRank, the number of iterations will not change when the arithmetic, including both its precision and order, remains unchanged. Although numerical error propagation could cause minor variations, comparison of Tables 6 and 7 reveals no difference in the required number of iterations for our algorithms and datasets.

In terms of the execution speed, including both the end-to-end runtime and the time per iteration, there is no instance where the global variant outperforms the local variant. In terms of the end-to-end runtime, the state-of-the-art PageRank implemented directly on top of ALP/GraphBLAS is fastest in eight cases, which are exactly the ones where the canonical PageRank achieves the fastest time per iteration. However, there is one exception where the local variant does achieve the fastest end-to-end runtime, even though in terms of the time per iteration the baseline outperforms the local variant. Indeed, differences in the time per iteration are more pronounced between the state-of-the-art baseline and the local variant, with speedups ranging 0.403–10 times.

Table 8 collects the speedups for all variants, similar to the SCC study in Section 6.3. The speedups ALP/Pregel achieves on bundle_adj (9.24-11.0x) are significantly higher than the speedups achieved by the baseline (1.81x). This is due to the adversarial structure of the graphs, which groups high-degree vertices in one cluster. Since the ALP/GraphBLAS baseline employs dense unmasked vectors only, the shared-memory parallel ALP backend reverts to static scheduling during SpMV multiplication. The ALP/Pregel variants, however, use a sparse structural mask for which the backend reverts to dynamic scheduling during SpMV multiplication. This, in turn, avoids the imbalance induced by the adversarial graph structure.

The ALP/GraphBLAS PageRank algorithm generally achieves better speedups than the ALP/Pregel variants, and the global variant tends to achieve better speedups than the local variants. While all implementations rely on the same OpenMP-enabled ALP/GraphBLAS backend, they rely on different OpenMP scheduling: as discussed earlier, the ALP/GraphBLAS variant employs static scheduling during sparse matrix–vector multiplication, whereas the ALP/Pregel variants employ dynamic scheduling. Additionally, ALP/Pregel employs dynamic scheduling while executing the vertex-centric program through the element-wise lambda. This is because the user-defined lambda may induce varying workload, depending on the index on which it operates. Moreover, some dense vector–vector operations in the baseline revert to static scheduling, while the same operations in the ALP/Pregel variants are part of the dynamically scheduled element-wise lambda. Some ALP/Pregel metadata updates performed via level-1 operations in the global variant also employ static scheduling, whereas in the local variants they revert to dynamic scheduling.

Another major effect is regarding the workspace of the ALP/Pregel programs. Recall that executing any Pregel program requires eight vectors, whereas the ALP/GraphBLAS PageRank implementation in Algorithm 3 requires only four. This translates to a benefit for smaller graphs, as the baseline implementation induces higher data reuse for the same problems within the same limited caches. This is compounded by an explicit materialization of the messages between vertices in the form of incoming and outgoing messages, meaning that computed values are copied twice as many times as in the baseline implementation, which holds zero such copies.

Algorithm 3 A simplified representation of the ALP/GraphBLAS PageRank

```
using namespace grb;
void pagerank(
    Vector< double > &pr, const Matrix< NonzeroT > &L,
    Vector< double > &temp, Vector< double > &pr_buffer,
    Vector< double > &row_sum,
    const double alpha,
    const double conv
) {
    Monoid< operators::add< double >, identities::zero > addM;
    Semiring< operators::add< double >, operators::mul< double >,
        identities::zero, identities::one > realRing;
    const size_t n = nrows( L );
    set( temp, 1 );
    set( row_sum, 0 );
    vxm< descriptors::dense | descriptors::transpose_matrix >( row_sum, temp, L, realRing );
    eWiseLambda( [ &row_sum, &alpha, &zero ]( const size_t i ) {
            if( row_sum[ i ] > 0 ) { row_sum[ i ] = alpha / row_sum[ i ]; }
        }, row_sum );

    double dangling, residual;
    do {
        residual = dangling = 0;
        foldl< descriptors::invert_mask >( dangling, pr, row_sum, addM );
        set( temp, 0 );
        eWiseApply( temp, pr, row_sum, operators::mul< double >() );
        dangling = ( alpha * dangling + 1 - alpha ) / static_cast< double >( n );
        set( pr_buffer, 0 );
        vxm( pr_buffer, temp, L, realRing );
        foldl< descriptors::dense >( pr_buffer, dangling, addM );
        dot< descriptors::dense >( residual, pr, pr_buffer, addM, operators::abs_diff< double >() );
        if( residual <= conv ) { break; }
        std::swap( pr, pr_buffer );
    } while( true );
}
```

Table 7 Results, in milliseconds, of executing shared-memory parallel PageRank algorithms. Like in Table 6, this table compares the ALP/Pregel variants against the baseline shared-memory parallel ALP/GraphBLAS variant.

ALP/Pregel				ms. per Iteration		
Dataset	**Global**	**Local**	**Baseline**	**Global**	**Local**	**Baseline**
1	31.1 (40)	29.2 (39)	*37.6* (52)	0.778	0.749	**0.723**
2	43.3 (43)	37.0 (41)	*53.8* (52)	1.01	**0.902**	1.03
3	58.8 (38)	38.9 (36)	29.0 (48)	1.55	1.08	**0.604**
4	280 (66)	*224* (51)	1290 (60)	4.24	**4.39**	21.5
5	157 (31)	127 (33)	35.1 (43)	5.06	3.85	**0.816**
6	*203* (31)	*62.3* (34)	32.1 (30)	6.55	1.83	**1.07**
7	263 (36)	163 (36)	*93.0* (45)	7.31	4.53	**2.07**
8	412 (40)	272 (33)	*146* (44)	10.3	8.24	**3.32**
9	367 (38)	243 (36)	87.7 (48)	9.66	6.75	**1.83**
10	1170 (35)	333 (35)	701 (46)	33.4	**9.51**	15.2
11	2440 (103)	878 (96)	5030 (55)	23.7	**9.15**	91.5
12	11500 (115)	4420 (104)	2750 (73)	100	42.5	**37.7**
13	9800 (78)	7560 (72)	2680 (78)	126	105	**34.4**

Table 8 Speedups of the parallel PageRank variants from Table 7 versus the sequential variants from Table 6.

Speedup vs. Itself				Speedup vs. Baseline	
Dataset	**Global**	**Local**	**Baseline**	**Global**	**Local**
1	1.12	0.846	0.835	1.21	**1.29**
2	4.43	3.19	3.66	1.24	**1.45**
3	2.98	2.03	3.10	0.493	0.746
4	11.0	9.24	1.81	4.61	**5.76**
5	4.50	3.59	12.4	0.224	0.276
6	4.81	1.02	11.7	0.158	0.515
7	10.8	7.24	31.8	0.354	0.571
8	12.4	9.19	32.5	0.354	0.537
9	5.34	4.06	12.5	0.239	0.361
10	17.8	8.35	35.7	0.599	**2.11**
11	16.6	13.0	3.60	2.06	**5.73**
12	13.3	10.4	26.2	0.239	0.622
13	8.94	7.78	23.2	0.273	0.354

The switch from static to dynamic scheduling, the loss of data locality, and the explicit materialization of messages result in a worst-case slowdown of 0.340×, comparing the baseline with global variants. Additional uses of dynamic instead of static scheduling in vector–vector and vector–scalar operations compound the worst-case slowdown to 0.212×, comparing the baseline with local variants. The effects of losing data locality should be reduced for larger problems.

Although these slowdowns indicate that ALP/Pregel programs scale worse than pure ALP/GraphBLAS ones, this may be resolved partially by investigating alternative dynamic scheduling techniques within ALP. However, the overheads incurred from copying messages and the loss of data reuse opportunities for small problems remain unavoidable. Nevertheless, the local variant leads to the fastest end-to-end shared-memory parallel execution in 5 out of 13 cases, with speedups of up to 5.76× compared to the baseline.

7 Conclusions

The paramount challenge in contemporary research on programming models and compilers lies in striking a balance between providing easy-to-use programming interfaces that make future highly parallel and heterogeneous compute systems accessible to humble programmers, and the necessity to:

- support an ever-increasing number of architectures,
- deploy over increasingly large-scale systems that involve multiple architectures, and
- deal with a mixture of shared- and distributed-memory connectivity across compute units.

In the interest of exposing humble programming models that are usable by the vast majority of programmers, a hit in performance and scalability may be acceptable. Ideally, however, humble code — once compiled — should perform on par with expert code, thus turning humble programmers into heroes. A recent study shows that the humble ALP/GraphBLAS outperforms the state-of-the-art frameworks for three graph and sparse matrix algorithms [MAY23]. ALP/Pregel exposes a vertex-centric programming model and translates vertex-centric programs into ALP/GraphBLAS at compile time, while offering benefits in terms of its shared- and distributed-memory auto-parallelization and other optimizations. This work demonstrates that a vertex-centric SCC algorithm implemented using ALP/Pregel obtains up to 17.5× speedup on a dual-socket Intel Xeon machine. Similarly, an auto-parallelized vertex-centric PageRank-like algorithm obtains speedups up to 17.8×, confirming that ALP/Pregel programs are able to scale. Furthermore, comparing the vertex-centric program against a highly optimized canonical PageRank algorithm results in faster sequential execution in 12 out of 13 cases, and speedups of up to 8.99×. In shared-memory parallel execution, the ALP/Pregel algorithm is fastest in 5 out of 13 cases, with speedups of up to 5.76×. These results confirm that humble programs may achieve hero-level performance, and demonstrate that novel approaches based on such humble programming interfaces may even outperform highly optimized canonical solutions.

7.1 Outlook

Different groups of humble programmers may prefer different humble programming interfaces. As such, the compute industry and researchers should strive to identify a set of humble programming models that together appeal to the vast majority of programmers. Furthermore, in an ideal scenario, only a few humble programming interfaces with corresponding optimized software stacks would be required. This paradox can be resolved by translating many useful humble programming interfaces into just a few optimized humble software stacks.

The reported speedups from the ALP/Pregel programs are achieved even though those programs are expanded into an ALP program, making use of its standard implementation. This not only validates the notion of supporting multiple humble programming models on top of a single software stack, but also demonstrates that this is possible without significant performance overheads. It also solves the software bottleneck as it frees hero programmers of addressing optimization and portability concerns for the few fundamental software stacks only — in this case, the ALP stack.

Even so, this paper does not argue that ALP should be the only fundamental programming model. It remains an open question as to how much of the general-purpose programming demands can be covered by ALP, as well as how many popular humble programming interfaces can be implemented on top of ALP without incurring significant performance overheads. Instead, this paper humbly argues that the future of software would be vastly improved if the industry and research communities explore more scalable humble programming interfaces, as well as identify the smallest possible subsets of foundational software stacks into which these humble programming models translate.

7.2 Improving ALP/Pregel

Pregel programs are naturally loosely coupled. At any point in time during execution, not all vertices are required to be executing the same round. Specifically, vertices whose round-i messages have been received can immediately continue with round $i+1$, regardless of whether other vertices have yet to execute the ith computation phase or whether they are waiting on incoming round-i messages. This insight can be applied recursively to realize pipelines of arbitrary length, bounded only by the underlying graph structure and the available memory for caching in-flight messages.

For example, a Pregel program running on the source vertex of a line graph can progress arbitrary round numbers ahead of other vertices. This leads to a pipeline depth bounded by the length of the graph. However, a Pregel program running on any vertex of a fully connected graph must execute in lock-step with all other programs due to being connected in an all-to-all fashion. While ALP/Pregel transforms the program into a series of level-1 operations separated by a message exchange driven by an SpMV multiplication, the use of a nonblocking implementation of ALP/GraphBLAS such as by Mastoras et al. [MAY23, MAY22] realizes overlap between the communication and computation phases of a single round. Introducing buffers for incoming messages in ALP/GraphBLAS may likewise enable overlapping rounds of vertex-centric computations. This will be the next step to take to enhance the ALP/Pregel performance. Such overlapping should be particularly effective on distributed-memory architectures, which, although supported by the current ALP/Pregel implementation, has not been evaluated in this work. Such comparison will be included in our future work, in which we will also compare the effects of loosely coupled execution. However, this requires bringing the nonblocking ALP/GraphBLAS capabilities to the distributed-memory parallel case. From the single-node perspective, a nonblocking variant that exploits the underlying graph structure in order to eliminate the need for caching incoming messages — thereby saving memory resources and increasing data reuse — seems a most promising direction.

7.3 Beyond ALP/Pregel

While this paper shows that realizing the Pregel humble programming model on top of ALP is possible and thus may benefit from the high performance and scalability guaranteed by ALP, not all algorithms and workloads are amenable to either pure ALP or vertex-centric programming. We need to expand the range of humble models that we can automatically translate into ALP. For MapReduce, for example, the following insights may automatically translate MapReduce programs into ALP:

- a one-to-one map phase corresponds to the element-wise lambda function;
- one-to-none or one-to-many maps may be realized by the parallel I/O mode introduced by ALP/GraphBLAS [YDNNS20]; and
- a reduce phase may be realized by a four-step process:
  (1) unzipping a key-value ALP/GraphBLAS vector of length n into two vectors of n keys and n values,
  (2) zipping the two vectors into an ALP/GraphBLAS matrix of size $k \times n$,
  (3) performing the reduction via an SpMV multiplication, and
  (4) transforming the resulting vector of values back into a vector of key-value pairs.

This ALP/MapReduce approach is currently under implementation and will be evaluated and released as the next additional humble programming interface to ALP. Inspirations to further humble interfaces on top of ALP include NumPy [ADH+01] and Spark [ZCF+10] as other examples of high-impact humble programming models, as well as programming models based on set algebra [BVSS+21, BKK+21] and commutative sets [KPW+07, PGZ+11].

References

[ADH+01] David Ascher, Paul F Dubois, Konrad Hinsen, Jim Hugunin, Travis Oliphant, *et al.*, "Numerical Python," 2001.

[AHU74] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman, "The design and analysis of computer algorithms," *Reading*, *MA*, 19(4), 1974.

[Alg21a] Algebraic Programming. ALP/GraphBLAS release v0.6. https://github.com/Algebraic-Programming/ALP, 2021. Last retrieved on April 6th, 2023.

[Alg21b] Algebraic Programming. ALP/GraphBLAS release v0.6. https://gitee.com/CSL-ALP/graphblas, 2021. Last retrieved on April 6th, 2023.

[AS87] Baruch Awerbuch and Yossi Shiloach, "New connectivity and MSF algorithms for shuffle-exchange network and PRAM," *IEEE Transactions on Computers*, 36(10):1258–1263, 1987.

[Ave11] Ching Avery, "Giraph: Large-scale graph processing infrastructure on Hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[BBM+19] Benjamin Brock, Aydin Buluç, Timothy Mattson, Scott McMillan, and José Moreira, "The GraphBLAS C API specification," *GraphBLAS.org*, *Tech. Rep*, 2019.

[BKK+21] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pp. 282–297, New York, NY, USA, 2021.

[BMM+17] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium workshops (IPDPSW)*, pp. 643–652. IEEE, 2017.

[BVSS+21] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, Peter Tatkowski, Esref Ozdemir, Adrian Balla, Marcin Copik, Philipp Lindenberger, Marek Konieczny, Onur Mutlu, and Torsten Hoefler, "GraphMineSuite: Enabling high-performance and programmable graph mining algorithms with set algebra," *Proc. VLDB Endow.*, 14(11):1922–1935, Jul. 2021.

[CLR92] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest, "Introduction to Algorithms," MIT Press, first edition, 1992.

[Dav19] Timothy A Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra," *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[DG08] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, 51(1):107–113, 2008.

[DH11] Timothy A Davis and Yifan Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[Dij72] Edsger W Dijkstra, "The humble programmer," *Communications of the ACM*, 15(10):859–866, 1972.

[DS00] James C Dehnert and Alexander Stepanov, "Fundamentals of generic programming," *International Seminar on Generic Programming*, pp. 1–11, Springer, 2000.

[GJ+10] Gaël Guennebaud, Benoit Jacob, *et al.*, "Eigen Technical report," TuxFamily, 2010.

[GXD+14] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pp. 599–613, 2014.

[KG11] Jeremy Kepner and John Gilbert, "Graph algorithms in the language of linear algebra," SIAM, 2011.

[KJ18] Jeremy Kepner and Hayden Jananthan, "Mathematics of big data: Spreadsheets, databases, matrices, and graphs", MIT Press, 2018.

[KPW+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 211–222, 2007.

[LM11] Amy N Langville and Carl D Meyer, "Google's PageRank and beyond," Princeton university press, 2011.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 135–146, 2010.

[MAY23] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman, "Design and implementation for non-blocking execution in GraphBLAS: tradeoffs and performance," in *ACM Transactions on Architectures and Code Optimization* 20(1), Article 6:1-23, 2023.

[MAY22] Aristeidis Mastoras, Sotiris Anagnostidis, and A. N. Yzelman, "Nonblocking execution in GraphBLAS," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 230–233, IEEE, 2022.

[MWM15] Robert Ryan McCune, Tim Weninger, and Greg Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd, "The PageRank citation ranking: Bringing order to the web," Technical report, Stanford InfoLab, 1999.

[PGZ+11] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P Johnson, and David I August, "Commutative set: A language extension for implicit parallel programming," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 1–11, 2011.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, IEEE, 2010.

[SM09] Alexander A Stepanov and Paul McJones, "Elements of programming," Addison-Wesley Professional, 2009.

[SR14] Alexander A Stepanov and Daniel E Rose, "From mathematics to generic programming," Pearson Education, 2014.

[SV80] Yossi Shiloach and Uzi Vishkin, "An $\mathcal{O}(\log n)$ parallel connectivity algorithm," Technical report, Computer Science Department, Technion, 1980.

[SW14] Semih Salihoglu and Jennifer Widom, "Optimizing graph algorithms on Pregel-like systems," *Proceedings of the VLDB Endowment*, 7(7), 2014.

[SY19] Wijnand Suijlen and A. N. Yzelman, "Lightweight Parallel Foundations: a model-compliant communication layer," arXiv:1906.03196v1, 2019.

[Val90] Leslie G Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 33(8):103–111, 1990.

[YDNNS20] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen, "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation," 2020.

[ZAB20] Yongzhe Zhang, Ariful Azad, and Aydin Buluç, "Parallel algorithms for finding connected components using linear algebra," *Journal of Parallel and Distributed Computing*, 144:14–27, 2020.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

Asynchronous Training and MoRe: Momentum Reconstruction for Reduced Memory Model Parallel Pipeline Training

Roman Talyansky, Zach Melamed, Pavel Kisilev, Ido Hakimi

Abstract

Large neural network training with model parallel (MP) pipeline execution leads to asynchronous training with significant levels of gradient staleness. This poses high memory requirements that limit model size and therefore undermine the goal of MP execution. In this paper, we present an extended overview of the recent asynchronous training techniques and then propose the Momentum Reconstruction (MoRe) optimization technique, yielding an a-SlowMo family of algorithms, including SGD with momentum, Adam, and AdamW. This allows us to achieve the convergence rates and generalization properties of SotA ASGD methods, while reducing their memory requirements by a factor of two. In turn, we can unleash the potential of training larger models on the same hardware, while paying a small computational overhead. We provide experimental validation of our solution on large CNN and Transformer architectures, and this includes training GPT models.

Keywords

optimization, distributed optimization, model parallel pipeline training, synchronous SGD

1 Introduction

Deep neural networks (DNN) have solved problems in various application domains, including image [1, 2, 3], video [4, 5, 6], and language models [7, 8]. In the quest for higher accuracy, the number of DNN model parameters has grown tremendously in recent years, making DNN training an extremely time-consuming and resource-hungry task [9].

With the growing size of DNN models, the memory capacity of a training device, or worker, has become a limiting factor in solutions, where each device keeps an entire model replica [10, 11]. To relax the memory capacity constraint within pipelined parallelism, we partition the model layers into *stages*, and assign them to the workers [12, 10]. Each worker runs the forward and backward pass for the stage it is assigned to and sends layer output and propagated error for computing gradients to the next worker node in the pipeline for the forward and backward pass computation. This basically forms pipelined processing for computing the entire forward and backward pass within DNN training. When a DNN model grows in the depth dimension with more layers, more stages and worker nodes are allocated in the pipeline, eliminating the memory capacity per worker as a limiting factor in the pipelined training strategy.

To improve the pipelined training efficiency, a minibatch is split into several microbatches that are fed into the pipeline sequentially, enabling different workers to work on the forward or backward passes of different microbatches simultaneously [10]. In synchronous SGD (SSGD), all workers must be synchronized on every minibatch boundary for the optimization step. This leads to the pipeline flush [10], which in addition to being expensive also has a significant impact on worker node utilization.

Asynchronous SGD (ASGD) enables workers to compute gradients over different model versions and minibatches to train several iterations simultaneously and therefore spares the need of synchronization on the minibatch boundary. In this way, pipeline flushing is eliminated, considerably improving the worker node utilization.

While ASGD reduces the idle time of workers, it introduces two additional problems. The first is gradient staleness, which can impact the convergence rate and final test accuracy, thereby negating the benefits of improved worker node utilization [13, 14]. The second problem is caused by multiple model versions. Specifically, multiple gradients and optimizer states co-exist in the training system, increasing the overall memory footprint [14, 12].

In this paper, we propose a MoRe framework to develop an algorithm family for MP pipeline training. The algorithm family achieves the convergence rate and final testing accuracy of SotA algorithms and reduces their memory requirements by a factor of almost 2. To demonstrate our framework, we developed two popular algorithms: SGD with momentum and AdamW.

Next, we describe the structure of the paper. We start with the related work and then point out the motivation for our research and its contributions. Then, we describe the essence of our proposed method and detail our MoRe algorithm family. For each optimizer variant among SGD with momentum and AdamW, we describe its master-based asynchronous variant and then demonstrate how to turn it into an equivalent master-less MoRe MP pipeline memory-efficient algorithm. We conclude with experimental results, showing the merits of our solution.

2 Overview of the Synchronous and Asynchronous Training Techniques

2.1 Synchronous Methods

[15] provides a practical low-rank gradient compression technique called PowerSGD for efficient distributed training. PowerSGD works by approximating each layer in a model independently using a low-rank singular value decomposition (SVD). Previous gradient compression works such as [16, 17] are incompatible with NVIDIA Collective Communications Library (NCCL) AllReduce operations; therefore, their speedups vanish with a fast network and highly optimized communication backend. On the other hand, PowerSGD is fully compatible with NCCL AllReduce and thus reduces the communication time (including coding and decoding) by 54% for ResNet-18 on CIFAR-10. PowerSGD features open-source code and is fully integrated in the standard PyTorch library today.

Massive parallelism is required to complete large-scale training within a reasonable amount of time. Large batch training is the key enabler of massive parallelism, but it often compromises generalization performance. Adaptive batching is a key technique for training neural networks with a very large average batch size while balancing the generalization trade-off. Recent work focuses on utilizing the gradient noise variance information to adjust batch sizes through various rules and measurements. Due to the high cost of analyzing the gradient variance and the second order information, existing methods can adapt the batch size only at a coarse-grained level. Due to these issues, the batch size usually needs to be compromised and large-scale training methods that achieve SotA performance still heavily rely on manual tuning efforts. [18] provides a practical algorithm called SimiGrad for fine-grained adaptive batching that supports swift batch size adaption at the iteration level. The core component of SimiGrad is a novel lightweight gradient variance measurement that can capture the gradient variance change per iteration without expensive analysis. They theoretically prove why cosine similarity is a good measure for gradient variance. SimiGrad is open source and is available in PyTorch.

[19] presents Adasum, a new distributed gradient aggregation algorithm for large-scale training that attempts to emulate a sequential execution in parallel. Its basic idea involves combining the individual model updates from nodes, each obtained by running a (small) minibatch from a starting model, into an update that would result from successively running these nodes from the same starting model, as in sequential training. They theoretically derive a simple Adasum formulation, which works without any additional hyperparameters. One of the key aspects of Adasum is that it boosts orthogonal gradient settings, that is when the workers have orthogonal gradients. The paper theoretically justifies Adasum and empirically demonstrates that it has a faster convergence rate and can work with larger average batch sizes than SSGD. The authors evaluate Adasum on both computer vision (ResNet-50 on ImageNet) and NLP (BERT) tasks and demonstrate that it is compatible with various optimizers such as Momentum-SGD, Adam, and LAMB. They provide open-source code with support for multiple deep learning frameworks (PyTorch and TensorFlow). They also claim that Adasum has been used by their company (Microsoft) in production for over three years to train various models.

[20] presents a unified framework — Cooperative-SGD — for the design and analysis of local-update SGD algorithms. Cooperative-SGD subsumes existing communication-efficient SGD algorithms such as periodic-averaging (Local SGD), Elastic Averaging SGD (EASGD), and Decentralized SGD (DPSGD), providing convergence guarantees. Cooperative-SGD enables the design of new communication-efficient SGD algorithms that strike the best balance between reducing communication overhead and achieving fast error convergence with a low error floor. The core of the framework is a mixing matrix W that defines the communication connectivity between different workers. This mixing matrix can be quite flexible, and it supports auxiliary variables such as anchor parameters. It requires the magnitudes of all eigenvalues except the largest one to be strictly less than one. The framework theoretically derives the trade-off between synchronization frequency and the mixing matrix sparsity. The paper later validates this with empirical experiments on CIFAR-10. The framework derives a theoretical optimal hyper-parameter α for the EASGD algorithm, indicating the elasticity of the algorithm.

[21] proposes a new distributed algorithm — Leader SGD (LSGD) — that is heavily based on the previous EASGD algorithm. In LSGD, parameter updates rely on two factors: a regular gradient step and a corrective direction dictated by the currently best-performing worker (leader). They theoretically analyze LSGD under the strongly convex setting and claim that LSGD is communication efficient as only the leader needs to broadcast all of its parameters. They evaluate LSGD on matrix completion, CIFAR-10, and ImageNet. The LSGD algorithm also supports an asynchronous variation. In the LSGD setting, workers are divided into local groups based on the relative communication speed. The key concept of LSGD is that workers are pulled toward the local/global leader rather than the central mass. In LSGD, workers are pulled toward the local leader in every τ iterations and pulled toward the global leader in every τ_G iterations. Adjusting these synchronization frequencies allows LSGD to adjust the amount of exploration. On CIFAR-10, LSGD outperforms the EASGD and SSGD algorithms; however, on ImageNet, it fails to achieve high accuracy and is outperformed by SSGD.

2.2 Hybrid Sync-Async Methods

While synchronous methods are based on computing gradients from the same central point in the weight search space, more recent approaches advocate a new type of settings where gradients are computed from different points located in a small proximity of the weight search space. These settings include Local SGD [22], Decentralized SGD [23]], and Delayed Gradient Averaging (DGA) [24]. We refer to these settings as sampling staleness, as computing gradients from different points of the search space may be viewed as sampling gradients from different points, and it also resembles gradient staleness that emerges in ASGD solutions [25]. The main common goal behind sampling staleness approaches is to reduce the idle time of workers during gradient merging in SSGD solutions.

There are two camps in optimization literature. In most optimization approaches, sampling staleness is viewed as a noise, whose negative impact should be minimized. However, a few recent approaches show that sampling staleness, when managed deliberately, does not compromise the convergence rate and generalization, but instead, improves them over the single-worker baseline. In the following section, we review some work from both camps.

D-SGD paper

The first paper we cover is on consensus control in decentralized training [23]. In Decentralized SGD (D-SGD), as opposed to SSGD, nodes are only partially connected and several iterations are required to propagate a gradient from one node to all other nodes, as shown in Figure 1. In general, the higher the graph connectivity is, the fewer the iterations are needed to perform this gradient propagation. N nodes solve the following problem:

$$f^* = \min_{x \in R^d} f(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \ , \qquad (1)$$

where $f_i(x) = \mathbb{E}_{\xi \in D_i}[F_i(x,\xi)]$.

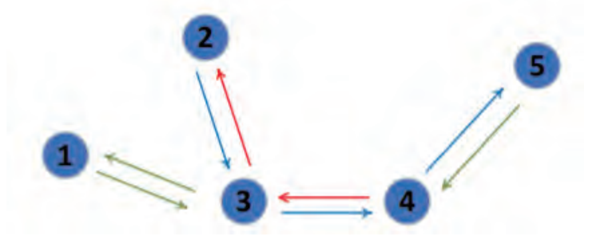


Figure 1 D-SGD example

Each worker $i \in [n]$ maintains local parameters $x_i^t \in R^d$ and updates them as

$$x_i^{t+1} = \sum_j = 1^n w_{i,j}(x_j^t - \eta \nabla F_j(x_j^t, \xi_j^t))$$

through a stochastic gradient step based on a sample, followed by gossip averaging with neighboring nodes in the network encoded by the mixing weights $w_{i,j}$.

We define $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, $X = [x_1,\ldots,x_n] \in R^{d\times n}$, and $\bar{X} = [\bar{x},\ldots,\bar{x}] = X\frac{1}{n}\mathbf{1}\mathbf{1}^T$.

The D-SGD analysis requires the following assumptions:

- Assumption 1: (Mixing matrix). Every sample of the (possibly randomized) mixing matrix $W = \{w_{i,j}\} \in R^{n\times n}$ is doubly stochastic and there is a parameter $p > 0$, s.t. $\mathbb{E}_W[||XW - \bar{X}||_F^2] \le (1-p)||X - \bar{X}||_F^2$ for any $X \in R^{d\times n}$.
- Assumption 2: (L-smoothness) $||\nabla f_i(x) - \nabla f_i(y)|| \le L||x-y||$.
- Assumption 3: (Bounded noise and diversity). There are constants σ^2 and ξ^2, s.t. for any $x_1,\ldots,x_n \in R^d$,

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\xi_i}[||\nabla F_i(x_i,\xi_i) - \nabla f_i(x_i))||_2^2] \le \sigma^2$$

and

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_{\xi_i}[||\nabla f_i(x_i) - \nabla f(x_i)||_2^2] \le \xi^2 .$$

Next, we point out the gap in the convergence rate between D-SGD and SSGD. To ensure

$$\frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}[||\nabla f(\bar{x}^t)||_2^2] \le \epsilon \ ,$$

D-SGD requires $O(\frac{\sigma^2}{\eta\epsilon^2} + \frac{\sqrt{p}\sigma+\xi}{p\epsilon^{3/2}} + \frac{1}{p\epsilon})$ steps for step size $\gamma \le O(\frac{p}{L})$, while SSGD requires $O(\frac{\sigma^2}{\eta\epsilon^2} + \frac{1}{\epsilon})$ steps for step size $\gamma \le O(\frac{1}{L})$.

To reduce the gap between the convergence rate of D-SGD and SSGD, we first introduce the consensus distance

$$\Xi_t^2 = \frac{1}{n}\sum_{i=1}^{n} ||\bar{x}^t - x_i^t||$$

and define Critical Consensus Distance (CCD)

$$\Gamma_t^2 = \frac{1}{Ln}\gamma\sigma^2 + \frac{1}{8L^2}||\nabla f(\bar{x}^t)||^2 .$$

The paper's analysis shows that if $\Xi_t^2 \le \Gamma_t^2$, we can choose a larger step size $\gamma \le O(\frac{1}{L})$ and recover the convergence rate of SSGD to $O(\frac{\sigma^2}{\eta\sigma^2})$.

These results conclude the analytical part of the paper. We then start the empirical validation of importance for CCD.

For CNN experiments, this paper chooses CIFAR-10 and ImageNet problems and the step decay learning rate (LR) schedule. Let Ξ_{max} be the typical consensus distance in an uncontrolled D-SGD experiment. Then, in a controlled experiment, one of the LR constant spans is chosen, and its consensus distance is set to $\alpha \cdot \Xi_{max}$, for $\alpha \in (0,1]$, e.g. $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$, etc., as shown in Figure 2. More gossip iterations are used to control the consensus distance. For the other spans, the experiments used SSGD, which ensures $\alpha = 0$.

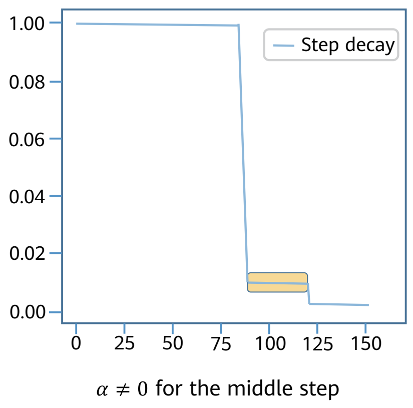


Figure 2 Controlled CCD experiment

The preceding experiments lead to the following observations. The initial LR span is the most critical one. Large consensus distances at the initial LR span may lead to generalization loss that cannot be recovered with more iterations. The preceding findings are consistent with the findings in Local SGD [26]. When a large consensus distance is allowed at the middle of LR spans, the accuracy of the CCD training surpasses the accuracy of a single-worker baseline. The last LR spans are mostly unaffected by consensus distance.

To conclude, our review of this paper confirms that it supports our intuition: If used in a controlled manner, sampling staleness is harmless and can improve generalization and convergence rate, compared to a single-worker baseline.

Local SGD paper

Next, we cover paper [22] from the realm of Local SGD. Within Local SGD, each worker performs τ local steps, as depicted in Figure 3, where τ is a parameter of the algorithm. The problem setting of this paper follows problem definition (1).

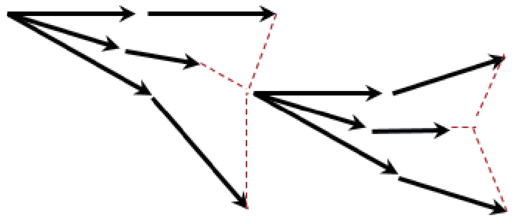


Figure 3 Local SGD

Next, we list the assumptions of this paper.

- Assumption 1 (Heterogeneity). $\{f_i\}_{i=1}^n$ is second-order ζ-heterogeneous, i.e., for any $i; i' \in [n]$,
$$||\nabla^2 f_i(x) - \nabla^2 f_{i'}(x)|| \leq \zeta,\ \forall x \in R^d\ .$$
- Assumption 2 (Smoothness). For any $i \in [n]$ and $\xi \in supp(D_i), \ell(\cdot, \xi)$ is L-smooth, i.e.,
$$||\nabla \ell(x, \xi) - \nabla \ell(y, \xi)|| \leq L||x - y||, \forall x, y \in R^d\ .$$
- Assumption 3 (Existence of global minimum). f has a global minimizer $x_* \in R^d$.
- Assumption 4 (Bounded gradient variance). For every $p \in [P]$,
$$\mathbb{E}_{z\sim D_p}||\nabla \ell(x, \xi) - \nabla f_p(x)2||^2 \leq \sigma^2\ .$$

Note that the authors of this paper point out that Assumption 2 implies $\xi \leq 2L$, i.e., the heterogeneity is bounded by the smoothness. The paper views sampling staleness as noise and examines four techniques to reduce its negative impact: localization, bias reduction, stochastization, and variance reduction. Next, we discuss each technique separately.

- Localization. In local methods, each worker independently optimizes the local objective and periodically communicates the current solution, as shown in Figure 3. To an extent, the local gradient $\nabla f_i(x)$ can be regard as a biased estimator of the global gradient $\nabla f(x)$. One of the limitations of Local SGD is the existence of the local gradient's $\nabla f_i(x)$ potential bias to approximate the global one $\nabla f(x)$ for heterogeneous local objectives $\{f_i\}_{i=1}^n$. The introduced bias $||\nabla f_i(x) - \nabla f(x)||$ does not usually converge to zero as $x \to x_*$.
- Bias reduction. The paper introduces a bias reduction technique for the local gradient. Concretely, the local estimator is constructed $\nabla f_i(x) - \nabla f_i(x) + \nabla f(x_0)$ to approximate $\nabla f(x)$. Here, x_0 is the previously communicated solution. Under the second order ζ-heterogeneity, the bias can be bounded as $||\nabla f_i(x) - \nabla f_i(x_0) + \nabla f(x_0) - \nabla f(x)|| \leq \zeta||x - x_0||$, meaning that the bias converges to zero as x and $x_0 \to x_*$. Therefore, the bias of the introduced estimator is reduced by utilizing the periodically computed global gradient $\nabla f(x)$.
- Stochastization. To avoid expensive computation of the gradient over the entire training set, gradient $\nabla \ell(x, \xi)$ can be computed over a single sample $\xi \sim D_i$ to approximate $\nabla f(x) = \mathbb{E}_{z\sim D}[\nabla \ell(x, \xi)]$. While stochastization reduces the computational cost of gradient computation, it leads to higher variance, affecting convergence speed. Stochastization can be introduced into the bias-reduced estimator as $\ell(x, \xi) - \ell(x_0, \xi) + (1/n)\sum_{i=1}^n \ell(x_0, \xi_{i'})$, where $\xi \sim D_i, \xi' \sim D_{i'}$ for $i' \in [n]$.
- Variance reduction. The authors of this paper introduce a variance reduction technique to reduce the variance of the gradient estimator due to local gradient estimation. The essence of variance reduction is utilization of the periodically computed full gradient $\nabla f(x)$. In non-distributed cases, a variance-reduced estimator is defined as $\nabla \ell(x, \xi) - \nabla \ell(x_0, \xi) + \nabla f(x_0)$ with $\xi \sim D$. The estimator is unbiased, and its variance $\mathbb{E}_{\xi\sim D}||\nabla \ell(x, \xi) - \nabla \ell(x_0, \xi) + \nabla f(x_0) - \nabla f(x)||^2$ can be bounded by $L^2||x - x_0||^2$, where L is the smoothness parameter of ℓ. If x and $x_0 \to x_*$, the variance converges to zero. Therefore, the estimator reduces the variance

caused by stochastization and maintains computational efficiency by using periodically computed global full gradients. Next, the authors of this paper show an algorithm that uses variance reduction twice: once prior to the local steps and then at the local steps. The algorithm runs on a single-layer neural network architecture and CIFAR-10 dataset. The authors of this paper show that their method is first to break the barrier of the $1/\epsilon$ barrier of communication complexity, when the local computation budget is $\to \infty$.

Algorithm 2.1: DGA

```
Initialize: each worker with $x^i_{1,1} = x_1$ for $i \in [N]$, the number of local updates $K$, and the delayed parameter $D \geq 1$. Define $s = (D-1)//K$ as the integer quotient.
1:  for rounds $t = 1, ..., T$ do
2:    for client $i$ in parallel do
3:      Set $x^i_{t,1} = x^i_{t-1,K+1}$ as the last iterate at round $(t-1)$
4:      for $k = 1, \dots, K$ do
5:        Sample the stochastic gradient $g^i_{t,k}$ at the previous iterate $x^i_{t,k}$ and update
6:        if $k \neq D(\mod K)$ or $t-1-s < 1$ then
7:          $x^i_{t,k+1} = x^i_{t,k} - \eta g^i_{t,k}$
8:        else if $k = D(\mod K)$ and $t-1-s \geq 1$ then
9:          $x^i_{t,k+1} = x^i_{t,k} - \eta(g^i_{t,k} - m^i_{t-1-s} + \bar{m}_{t-1-s})$
10:       end if
11:       where $m^i_{t-1-s}$ is the accumulated gradient (see line 13) at the earlier round $t-1-s$, $\bar{m}_{t-1-s}$ is the average of $m^i_{t-1-s}$ among all clients, i.e., $\bar{m}^i_{t-1-s} = \frac{1}{N}\sum_i m^i_{t-1-s}$.
12:     end for
13:     Send the $t$-th round accumulated gradient $m^i_t = \sum_{k=1}^K g^i_{t,k}$ to all other clients.
14:   end for
15: end for
16: return $\bar{x}_T = \frac{1}{N}\sum_{i=1}^N x^i_{T,K+1}$
```

DGA paper

Next, we turn to delayed gradient techniques [24]. One of the pains of synchronous training is worker idle time during gradient merging, especially in federated learning scenarios. Delayed gradient approaches reduce this pain by overlapping gradient computation and gradient merging or averaging.

The problem setting of this paper also follows problem definition (1). Let τ_g be the time needed to compute a single gradient, and τ_c be the communication time, and $D = \lceil \frac{\tau_c}{\tau_g} \rceil$ be normalized communication cost — one communication round takes the same time as D gradient updates.

Next, we describe the assumptions of the paper.

- Assumption 1 (L-smoothness). $||\ell(x,\xi) - \ell(y,\xi)|| \leq L||x-y||$.
- Assumption 2 (Bounded gradients and variances). $\mathbb{E}||\nabla \ell_i(x,\xi)||^2 \leq G$ and $\mathbb{E}||\nabla \ell_i(x,\xi) - \nabla f_i(x)||^2 \leq \sigma^2$.

To simplify the description of the paper, we assume that the number of local steps K is equal to D, i.e., $K = D$. Now, we describe the DGA algorithm in Algorithm 2.1. To hide the communication latency, at each round, DGA in parallel allows each worker to compute K gradients, advancing local weights along the path of K gradients, and aggregates the paths of length K from each worker of the previous round.

Note that when the aggregated paths from the previous round arrive, each worker applies the *variance reduction* technique from [22] in line 9 to reduce the variance of using a smaller local minibatch.

Next, the authors of this paper analyze the convergence rate of DGA and prove the following result:

Corollary 2.3.1. When the function f is lower bounded with $f(w_1) - f^* \leq \Delta$ and the number of rounds T is large enough such that $T > N/(K+D)$, then set the step size $\eta = \frac{\sqrt{N}}{L\sqrt{T(K+D)}}$ yields

$$\frac{1}{TK}\sum_{t=1}^{T}\sum_{k=1}^{K}\mathbb{E}[||\nabla f(\bar{x}_{t,k})||^2] = O\left(\frac{2L\Delta + \sigma^2}{\sqrt{NTK}} \cdot \sqrt{1 + \frac{D}{K}} + \frac{N(K+D)}{T}\right).$$

Note that this result is better than the results of SotA FedAvg algorithm by a factor of $\frac{K+D}{K}$.

Finally, the authors of this paper show, via experiments on CIFAR-10 and ImageNet, that they can withstand delays of up to 20 gradients without significant loss in test accuracy, compared with the completely synchronous counterpart.

3 MoRe Related Work

3.1 MP Pipeline Variants

In this section, we summarize the main ideas of pipelined training, which are related to our setting. [12] proposed

PipeDream — an efficient asynchronous pipeline training algorithm. PipeDream uses *weight stashing* and *vertical sync*, which result in asynchronous updates of weights with a constant staleness above one. [27] proposed PipeDream-2BW — a memory-efficient pipeline training algorithm. It assumes the number of microbatches per minibatch is at least the same as the number of stages. [27] uses this assumption to develop a double-buffered weights scheme to ensure a small memory footprint and maintain the gradient staleness at 1. PipeMare [28] adapts the model weights and reduces the learning rate in the subsequent pipeline stages. On the one hand, this reduces the memory requirements. On the other hand, it may lead to a slower convergence rate.

3.2 ASGD Methods for MP Pipeline Training

In this section, we summarize the ASGD algorithms used for pipeline training. The Delay Compensation method [29] obtains a stale gradient, and uses it to approximate the gradient at the current iterate and make the optimization step. At a high scale, the approximation quality drops, leading to poor generalization. Staleness-aware ASGD [13] penalizes a gradient by its staleness. Since staleness is not always the right penalty measure, generalization with this method rapidly degrades as the number of workers increases. DANA-GA [25] uses a combination of parameter prediction and gradient penalization using a *gap* as a more appropriate penalization measure. While this method excels at good generalization, even at a high scale, it uses SGD with the momentum optimizer for each worker, needing two buffers per worker to store weights and momentum with the total memory requirement of at least $2N$ buffers for N workers.

4 Motivation

4.1 High Levels of Staleness

As we mentioned, [27] assumes the number of microbatches per minibatch is at least the same as the number of stages. [27] uses this assumption to develop a memory-efficient solution. In this section, we show that the continuously increasing model depth and limitations on minibatch and microbatch sizes lead to the setting in which the above assumption of [27] does not hold, i.e. setting, where there are fewer microbatches per minibatch than stages, making the solution of [27] inapplicable.

First, in large-scale DNN training, the minibatch size is limited from above, since minibatches that are too large may affect the convergence rate [30]. On the other hand, for efficient processing, the microbatch size is limited from below. The preceding upper bound on the minibatch size and lower bound on the microbatch size lead to an upper bound on the number of microbatches per minibatch.

Next, we examine a lower limit on the number of microbatches to fill in the pipeline. In pipelined training, each microbatch traverses each stage twice for forward and backward passes, before a gradient is computed for all the stages [10, 27]. For this reason, to minimize the pipeline bubbles, at least twice as many microbatches as stages should be concurrently processed in the pipeline [27].

As very large DNN models are likely to be split into more stages, more microbatches are required to fill in the pipeline. With a limited number of microbatches per minibatch, we need more minibatches to fill in the pipeline, leading to a setting where there are fewer microbatches per minibatch than stages and the staleness increases to above one. In this work, we focus on providing a solution for this setting.

4.2 High Memory Requirements

As we focus on a setting with a high level of staleness, we need an asynchronous solution to cope with this staleness without compromising test accuracy. As demonstrated in section 3.2, DANA-GA [25] is the only asynchronous solution that meets this demand but with high memory requirements, limiting the affordable model size and therefore compromising the goal of MP pipeline execution when training very large models. Table 1 summarizes the memory requirements of DANA-GA.

5 Contributions

DANA-GA [25] is the SotA in ASGD training. However, its memory requirements are considerably high — at least $2N$ memory buffers for N workers. In this work, we develop MoRe — a family of ASGD algorithms for pipelined training that achieve a better trade-off than DANA-GA. As demonstrated in section 8, for the SGD with momentum optimizer, the MoRe framework significantly reduces the memory requirements without compromising the convergence rate and final test accuracy. For the AdamW optimizer, MoRe improves the final test accuracy with only a minor increase in memory requirements, compared to DANA-GA. Also, Table 1 shows that MoRe and DANA-GA have a similar computational overhead.

6 Proposed Method

6.1 Momentum Reconstruction Foundation

In DANA-GA [25], each worker manages two buffers: for the parameters and the momentum, leading to a high memory demand. To cope with this problem, we developed the Momentum Reconstruction (MoRe) technique, which allows us to compute the value of per-worker momentum from the parameters of two workers. Therefore, if the algorithm stores per-worker parameters, it can compute the value of the per-worker momentum on the fly, eliminating the need of the momentum buffer. For this reconstruction, MoRe relies on the round-robin order of worker updates.

6.2 Bridging over DP and MP Terminology

Note that the term *worker* has different meanings in data parallel (DP) and MP pipeline settings. In DP settings, a worker denotes a separate computational unit that stores a model replica and has access to training data for computing gradients. In the context of the MP pipeline execution, a worker is a set of devices responsible for computing the forward and backward passes of a *stage* — a part of the model. In this paper, we discuss ASGD algorithms in the context of the MP pipeline execution and associate a worker with a model version that the pipeline manages at a certain time. In other words, the term *worker*, in our context, refers to a logical entity associated with a model version that the pipeline uses to compute a gradient. Let us consider Figure 4, where w_i is worker i, M_j is model j, and t_k is time k. When at times t_{12} and t_{15}, the pipeline starts the forward pass on model M_0, using microbatches 5 and 6. We associate worker w_3 with this model. In DP terminology, we say that worker w_3 starts the forward pass at times t_{12} and t_{15} on microbatches 5 and 6, finishes the backward pass, and merges the resulting gradient into model M_2 at time t_{26}.

6.3 Simulating MP Pipeline Execution

Let L denote the loss function and $\boldsymbol{x}_t^{(i)}$ denote the model parameters of worker i, which were assigned to this worker at time t. We assign worker indexes i as iteration t modulo the number of workers, i.e., $i = t \mod (N)$. Let us recall that according to our definition of a worker in the pipeline setting, the pipeline with N workers manages at each moment N model versions. We arrange them in a cyclic buffer of length N. Figure 5 illustrates this representation. The pipeline updates the workers' parameters in a round-robin manner. According to the MP pipeline execution model, at iteration t, worker i completes computing gradient $\boldsymbol{g}_{t-N}^{(i)} = \nabla L(\boldsymbol{x}_{t-N}^{(i)}; \xi_{t-N})$ over its parameters $\boldsymbol{x}_{t-N}^{(i)}$ and minibatch ξ_{t-N} and uses it to update the up-to-date model in the pipeline, i.e., $\boldsymbol{x}_{t-1}^{(i-1)}$, resulting in the following updated parameters of worker i:

$$\boldsymbol{x}_t^{(i)} = \boldsymbol{x}_{t-1}^{(i-1)} - f(\boldsymbol{g}_{t-N}^{(i)}), \qquad (2)$$

where f represents any optimization technique, such as momentum, Adam, etc. As gradient $\boldsymbol{g}_{t-N}^{(i)}$ is computed on parameters $\boldsymbol{x}_{t-N}^{(i)}$ and merged into $\boldsymbol{x}_{t-1}^{(i-1)}$, the staleness in MP pipeline computation with N workers is $N-1$.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}	t_{24}	t_{25}	t_{26}	t_{27}	t_{28}	t_{29}	t_{30}
	M_0														M_1												M_2				
	W_1	W_1	W_2	W_2									W_3	W_1	W_1	W_3			W_1	W_2	W_2	W_1			W_2	W_3	W_3	W_2			W_3
stage 1	1	2	3	4							B1	B1	5	B2	B2	6	B3	B3	7	B4	B4	8	B5	B5	9	B6	B6	10	B7	B7	11
stage 2		1	2	3	4				B1	B1		B2	B2	5	B3	B3	6	B4	B4	7	B5	B5	8	B6	B6	9	B7	B7	10	B8	B8
stage 3			1	2	3	4	B1	B1		B2	B2		B3	B3	5	B4	B4	6	B5	B5	7	B6	B6	8	B7	B7	9	B8	B8	10	B9
stage 4				1	B1	B1	2	B2	B2	3	B3	B3	4	B4	B4	5	B5	B5	6	B6	B6	7	B7	B7	8	B8	B8	9	B9	B9	10

Figure 4 Pipeline execution example with four stages

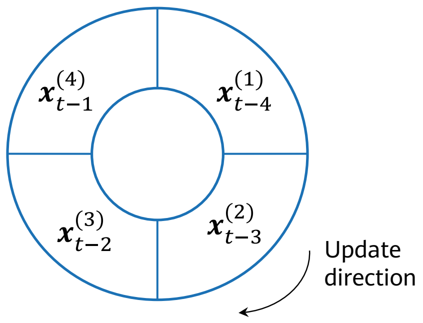


Figure 5 Cyclic execution in the pipeline with the current update $\boldsymbol{x}_t^{(1)} = \boldsymbol{x}_{t-1}^{(4)} - f(\boldsymbol{g}_{t-N}^{(i)})$

7 a-SlowMo Algorithm Family

SlowMo [31] is a synchronous framework that allows unification of numerous existing SotA training algorithms, e.g., Lookahead Optimizer [32] and Local SGD [33]. Based on the SlowMo foundation, we develop a new family of ASGD algorithms.

7.1 a-SlowMo

We start by turning SlowMo, Algorithm A.1, into its asynchronous variant (a-SlowMo) in Algorithm 7.1. To achieve this goal, we remove the workers' synchronization after the inner loop is over.

DANA-GA uses prediction of the master parameters, based on the sum of the per-worker momentum, for staleness mitigation. Our a-SlowMo variant achieves the same goal using the slow momentum, making a-SlowMo particularly suitable for asynchronous training.

Algorithm 7.1: Asynchronous SlowMo: a-SlowMo

Input: Base optimizer with learning rate γ_t; Inner loop steps τ; Slow learning rate α; Slow momentum factor β; Number of worker nodes m. Initial point $\boldsymbol{x}_{0,0}$ and initial slow momentum buffer $\boldsymbol{u}_0 = \mathbf{0}$.

```
1: for $t \in \{0, 1, ..., T-1\}$, where worker $i = t \mod (N)$ do
2:     Reset/Maintain/Average base optimizer buffers of worker $i$
3:     for $k \in \{0, 1, ..., \tau - 1\}$ do
4:         Base optimizer step: $\boldsymbol{x}_{t,k+1}^{(i)} = \boldsymbol{x}_{t,k}^{(i)} - \gamma_t \boldsymbol{d}_{t,k}^{(i)}$
5:     end for
6:     Update master: $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \alpha(\boldsymbol{x}_{t,\tau}^{(i)} - \boldsymbol{x}_{t,0}^{(i)})$
7:     Update slow momentum: $\boldsymbol{u}_{t+1} = \beta\boldsymbol{u}_t + \frac{1}{\gamma_t}(\boldsymbol{x}_{t,0}^{(i)} - \boldsymbol{x}_{t,\tau}^{(i)})$
8:     Update outer iterates of worker $i$: $\boldsymbol{x}_{t+1,0}^{(i)} = \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1}$
9: end for
```

7.2 a-SlowMo SGD with Momentum

Next, we specialize the general formulation of a-SlowMo, Algorithm 7.1, resulting in Algorithm 7.2. First, we assume that $\tau = 1$ and remove the inner loop. We follow DANA-GA and allow each worker to maintain its own momentum in the buffer $\boldsymbol{m}_t^{(i)}$ with $\boldsymbol{m}_t^{(i)} = \boldsymbol{d}_{t,0}^{(i)}$. For this reason, each worker manages two buffers: one for the model parameters and one for momentum.

We add weight decay in line 4 and gradient normalization in line 5, as we described in section 7.4. Note that we added the term $-\alpha(\gamma_{t-1} - \gamma_t\beta)\boldsymbol{u}_t$ to the master update rule in line 7, which does not show up in Algorithm 7.1. This term allows us to reconstruct the value of the per-worker momentum, as we show in Lemma 7.1.

Algorithm 7.2 uses the master weights and slow momentum, and it does not follow the pipeline execution model as in PipeDream. Next, we transform Algorithm 7.2 into an equivalent algorithm that follows this model.

Algorithm 7.2: a-SlowMo SGD with momentum

Input: Base optimizer with learning rate γ_t; Slow learning rate α; Slow momentum factor β; Fast momentum factor μ; Initial parameters $\boldsymbol{x}$ as initial master parameters and initial parameters of each worker, initial per worker momentum $\boldsymbol{m}_{t,k}^{(i)} \leftarrow \mathbf{0}$ and initial slow momentum buffer $\boldsymbol{u}_0 = \mathbf{0}$.

```
1: for $t \in \{0, 1, ..., T-1\}$, where worker $i = t \mod (N)$ do
2:     Base optimizer step:
3:     Compute gradient: $\boldsymbol{g}_t^{(i)}$ of worker $i$
4:     Add weight decay: $\boldsymbol{g}_t^{(i)} = \boldsymbol{g}_t^{(i)} + \lambda \boldsymbol{x}_{t+1-N}^{(i)}$
5:     $\boldsymbol{g}_t^{(i)} \leftarrow normalize\_gradient\left(\boldsymbol{g}_t^{(i)}\right)$
6:     Update momentum: $\boldsymbol{m}_t^{(i)} = \mu \boldsymbol{m}_{t-N}^{(i)} + \boldsymbol{g}_t^{(i)}$
7:     Update master: $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha\gamma_t \boldsymbol{m}_t^{(i)} - \alpha(\gamma_{t-1} - \gamma_t\beta)\boldsymbol{u}_t$
8:     Update slow momentum: $\boldsymbol{u}_{t+1} = \beta\boldsymbol{u}_t + \boldsymbol{m}_t^{(i)}$
9:     Update outer iterates of worker $i$:
       $\boldsymbol{x}_{t+1}^{(i)} = \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1}$
10: end for
```

7.3 Proposed MoRe SGD with Momentum

In this section we show how to shape Algorithm 7.2 into MoRe MP pipeline version. Lemma 7.1 shows how to reconstruct per-worker momentum.

Lemma 7.1. *(Proof in Appendix B) In Algorithm 7.2*

$$\boldsymbol{m}_t^{(i)} = -\frac{1}{2\alpha\gamma_t}(\boldsymbol{x}_{t+1}^{(i)} - \boldsymbol{x}_t^{(i-1)}). \tag{3}$$

Next, Lemma 7.2 allows us to shape Algorithm 7.2 into the MP pipeline form. It shows a recursive expression to compute the parameters of worker i at iteration $t+1$ from the parameters of worker $i-1$ at iteration t.

Lemma 7.2. *(Proof in Appendix B) In Algorithm 7.2, lines 7, 8, and 9, we can replace the computation of* $\boldsymbol{x}_{t+1,0}^{(i)}$ *with the following equivalent recursive expression:*

$$\boldsymbol{x}_{t+1}^{(i)} = \boldsymbol{x}_t^{(i-1)} + \mu \frac{\gamma_t}{\gamma_{t-N}}(\boldsymbol{x}_{t-N+1}^{(i)} - \boldsymbol{x}_{prev}) - 2\alpha\gamma_t \boldsymbol{g}_t^{(i)}. \tag{4}$$

Now, we use Lemma 7.2 to re-write Algorithm 7.2 in an equivalent form that suits MP pipeline execution model, resulting in Algorithm 7.3. Note that Algorithm 7.3 does not operate with the master parameters and slow momentum. However, they are still used *implicitly* due to the equivalence of Algorithms 7.2 and 7.3.

```
Algorithm 7.3: MoRe SGD with momentum
Input: Base optimizer with learning rate $\gamma_t$; Slow learning rate $\alpha$; Fast momentum factor $\mu$; Initial point $\boldsymbol{x}_{0,0}$.
1: for $t \in \{0, 1, ..., T-1\}$, where worker $i = t \mod (N)$ do
2:     Compute gradient $\boldsymbol{g}_t^{(i)}$ of worker $i$
3:     Add weight decay: $\boldsymbol{g}_t^{(i)} = \boldsymbol{g}_t^{(i)} + \lambda \boldsymbol{x}_{t+1-N}^{(i)}$
4:     Reconstruct momentum of the previous round:
       $\Delta_{t-N} = -(\boldsymbol{x}_{t-N+1}^{(i)} - \boldsymbol{x}_{prev})$
5:     Store weights of worker $i$: $\boldsymbol{x}_{prev} = \boldsymbol{x}_{t-N+1}^{(i)}$
6:     $\boldsymbol{g}_t^{(i)} \leftarrow normalize\_gradient\left(\boldsymbol{g}_t^{(i)}\right)$
7:     Update outer iterates of worker $i$:
       $\boldsymbol{x}_{t+1}^{(i)} = \boldsymbol{x}_t^{(i-1)} - \mu \frac{\gamma_t}{\gamma_{t-N}} \Delta_{t-N} - 2\alpha\gamma_t \boldsymbol{g}_t^{(i)}$
8: end for
```

Memory analysis

Algorithm 7.3 uses one memory buffer to store the previous parameters in line 5. Procedure $normalize\_gradient$ uses two buffers, as shown in section 7.4, leading to three buffers for the entire algorithm. In addition, each worker uses a single buffer to store weights. Therefore, Algorithm 7.3 requires $N+3$ buffers for N workers.

7.4 Gradient Burst Problem and its Solution with Normalization

In ASGD settings, we can observe short growth bursts in a certain entry of gradients along consecutive optimization iterations. Figure 6 shows such a burst at entry j. Gradient $\boldsymbol{g}_{t-\tau}^{(i)}$ is computed at time $t-\tau$, at the burst, and merged at time t, when the burst is over. Therefore, at time t, the value of the gradient at entry j significantly different from its value at time $t-\tau$, which can lead to optimization instability.

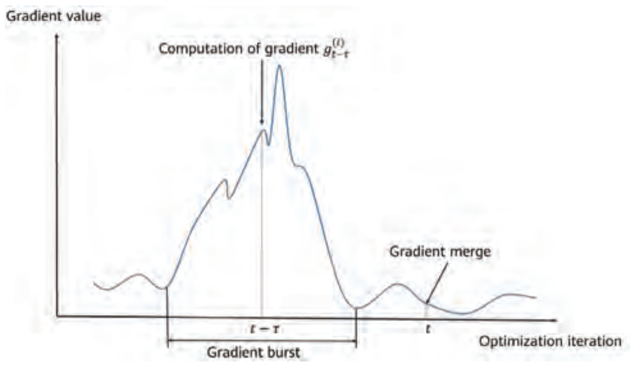


Figure 6 Burst of gradient growth

To cope with the burst problem, we propose gradient normalization, as outlined in Algorithm 7.4. When the momentum factor β_1 of the second moment $\boldsymbol{z}_t$ is much larger than the momentum factor β of the normalization momentum $\boldsymbol{v}_t$, e.g., $\beta_1 = 0.999$ and $\beta = 0.9$, the second moment $\boldsymbol{z}_t$ is almost unaltered by a short burst, while the normalization momentum $\boldsymbol{v}_t$ might be amplified by it. In this case, entry j of the normalization vector V is large; therefore, it normalizes the outdated value at entry j of gradient $\boldsymbol{g}_t^{(i)}$.

```
Algorithm 7.4: Procedure: $normalize\_gradient$
Context; Learning rate $\gamma_t$ of the base optimizer; Normalization momentum factor $\beta$; Second moment factor $\beta_1$; Momentum $\boldsymbol{m}_{t-N}^{(i)}$ of worker $i$.
Input: Gradient $\boldsymbol{g}_t^{(i)}$ of worker $i$.
Output: Normalized gradient $\boldsymbol{g}_t^{(i)}$.
1: Update second moment: $\boldsymbol{z}_{t+1} \leftarrow \beta_1 \boldsymbol{z}_t + (1-\beta_1)(\boldsymbol{m}_{t-N}^{(i)})^2$
2: Compute the bias-corrected second moment:
   $\hat{\boldsymbol{z}}_{t+1} \leftarrow \boldsymbol{z}_{t+1}/(1-\beta_1^{t+1})$
3: Calculate normalization vector: $V = \frac{\gamma_t}{\gamma_0} \frac{|\hat{\boldsymbol{v}}_t|}{\sqrt{\hat{\boldsymbol{z}}_{t+1}+\epsilon}} + \mathbf{1}^d$
4: Normalize gradient: $\boldsymbol{g}_t^{(i)} = \frac{1}{V} \cdot \boldsymbol{g}_t^{(i)}$
5: Update normalization momentum: $\boldsymbol{v}_{t+1} = \beta \boldsymbol{v}_t + \boldsymbol{g}_t^{(i)}$
6: Update bias-corrected normalization momentum:
   $\hat{\boldsymbol{v}}_{t+1} \leftarrow \frac{\boldsymbol{v}_{t+1}}{1-\beta^{t+1}}$
7: Return normalized gradient: $\boldsymbol{g}_t^{(i)}$
```

7.5 a-SlowMo AdamW, Master-Based

In this section, we extend a-SlowMo to the Adam [34] and AdamW algorithms [35]. We start with adapting Algorithm 7.2 to AdamW, resulting in Algorithm 7.5.

Algorithm 7.3 reconstructs the momentum of a worker using linear operations. However, the Adam update rule involves element-wise division of the first moment by the second moment, which is not linear, breaking the basis of our momentum reconstruction. To cope with this non-linearity, we note that to estimate the second moment, the Adam algorithm uses a large constant, e.g., 0.999, leading to a very slow change in the second moment estimation. This allows us to freeze the value of the second moment estimation in line 8 for N consecutive iterations with a negligible effect on the convergence rate. In turn, the freezing enables us to reconstruct the first moment of a worker by falling back to linear operations.

Note that we use expressions of the form $\boldsymbol{m}_{t+1} = \mu \boldsymbol{m}_t + \boldsymbol{g}_{t+1}$ to compute moment estimations. In Appendix E, we follow [34] and show that we should use bias correction via division of the momentum by $\frac{1-\mu^t}{1-\mu}$.

Algorithm 7.5 uses master weights and slow momentum, and it does not follow the pipeline execution model as in PipeDream. In the next section, we transform Algorithm 7.5 into an equivalent algorithm that follows this model.

Algorithm 7.5: AdamW in a-SlowMo framework

Input: Base optimizer with learning rate γ_t; Slow learning rate α; Slow momentum factor β; Fast momentum factor μ; Initial parameters $\boldsymbol{x}$ of master and each worker; Initial per worker momentum $\boldsymbol{m}_t^{(i)} \leftarrow \mathbf{0}$; Initial slow momentum buffer $\boldsymbol{u}_0 = \mathbf{0}$. $[\boldsymbol{g}_t^{(i)}]^2$ indicates the element-wise square $\boldsymbol{g}_t^{(i)} \odot \boldsymbol{g}_t^{(i)}$. Constants $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$; Second moment estimate $\boldsymbol{y}_0 = \mathbf{0}$; Weight decay factor λ.

```
1:  for $t \in \{1, 2, ..., T\}$, where worker $i = t \mod (N)$ do
2:      Compute gradient: $\boldsymbol{g}_t^{(i)}$ of worker $i$
3:      $g_t^{(i)} \leftarrow normalize\_gradient\left(\boldsymbol{g}_t^{(i)}\right)$
4:      Update first moment estimate: $\boldsymbol{m}_t^{(i)} = \mu \boldsymbol{m}_{t-N}^{(i)} + \boldsymbol{g}_t^{(i)}$
5:      Update second moment estimate: $\boldsymbol{y}_t = \beta_2 \boldsymbol{y}_{t-1} + [\boldsymbol{g}_t^{(i)}]^2$
6:      Compute bias-corrected first moment estimate:
        $\widehat{\boldsymbol{m}}_t^{(i)} = \boldsymbol{m}_t^{(i)}(1-\mu)/(1-\mu^{\lceil t/N \rceil})$
7:      if $(t-1) \mod N = 0$ then
8:      Discretize bias-corrected second moment estimate:
        $\widehat{\boldsymbol{y}} = \boldsymbol{y}_t(1-\beta_2)/(1-\beta_2^t)$
9:      end if
10:     Compute ratio: $\boldsymbol{r}_t = \widehat{\boldsymbol{m}}_t^{(i)}/(\sqrt{\widehat{\boldsymbol{y}}} + \epsilon) + \lambda \boldsymbol{x}_{t,0}^{(i-1)}$
11:     Update master: $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha\gamma_t \boldsymbol{r}_t - \alpha(\gamma_{t-1} - \gamma_t \beta)\boldsymbol{u}_t$
12:     Update slow momentum: $\boldsymbol{u}_{t+1} = \beta \boldsymbol{u}_t + \boldsymbol{r}_t$
13:     Update outer iterates of worker $i$:
        $\boldsymbol{x}_{t+1}^{(i)} = \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1}$
14: end for
```

7.6 Proposed MoRe AdamW

Lemma 7.3 shows that in Algorithm 7.5, it is redundant to store momentum buffers since their values may be calculated from the values of per-worker parameters.

Lemma 7.3. *(Proof in Appendix B) In Algorithm 7.5*

$$\boldsymbol{m}_t^{(i)} = -\bar{V}_t \frac{1-\mu^{\lceil t/N \rceil}}{1-\mu}\left(\sqrt{\widehat{\boldsymbol{y}}} + \epsilon\right) \odot \left(\frac{1}{2\alpha\gamma_t}(\boldsymbol{x}_{t+1}^{(i)} - \boldsymbol{x}_t^{(i-1)}) + \lambda \boldsymbol{x}_t^{(i-1)}\right). \quad (5)$$

Lemma 7.4 allows computing the parameters of worker i at iteration $t+1$ from the parameters of worker $i-1$ at iteration t, shaping Algorithm 7.5 into MP pipeline form.

Lemma 7.4. *(Proof in Appendix B) In Algorithm 7.5, line 13, we can replace the computation of $\boldsymbol{x}_{t+1,0}^{(i)}$ with the following equivalent recursive expression:*

$$\boldsymbol{x}_{t+1}^{(i)} = \boldsymbol{x}_t^{(i-1)} + STEP_t \ ,$$

where

$$STEP_t = \mu a_t \frac{\sqrt{\widehat{\boldsymbol{y}}_{prev}} + \epsilon}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} \odot \left(-b_t \Delta_{t-N} + c_t \boldsymbol{x}_{t-N}^{(i-1)}\right) - d_t \frac{\boldsymbol{g}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} - c_t \boldsymbol{x}_t^{(i-1)} \ ,$$

$$a_t = \frac{\bar{V}_{t-N}}{\bar{V}_t}\frac{1-\mu^{\lceil (t-N)/N \rceil}}{1-\mu^{\lceil t/N \rceil}}, \quad b_t = \frac{\gamma_t}{\gamma_{t-N}} \ ,$$
$$c_t = 2\alpha\gamma_t\lambda \ , \quad d_t = \frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N \rceil}}$$

We use Lemma 7.4 to shape Algorithm 7.5 in an MP pipeline form, as in PipeDream, resulting in Algorithm 7.6.

Algorithm 7.6: MoRe Adam

Step: $STEP_t$ is computed according to Lemma 7.4.

Input: Base optimizer with learning rate γ_t; Fast momentum factor μ; Initial parameters $\boldsymbol{x}$ as initial master parameters and initial parameters of each worker; Initial slow momentum buffer $\boldsymbol{u}_0 = \mathbf{0}$. $[\boldsymbol{g}_t^{(i)}]^2$ indicates the element-wise square $\boldsymbol{g}_t^{(i)} \odot \boldsymbol{g}_t^{(i)}$. Constants $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$; Second moment estimate $\boldsymbol{y}_0 = 0$, previous discrete second moment estimate $\widetilde{\boldsymbol{y}}_{prev} = \mathbf{1}$.

```
1:  for $t \in \{0, 1, ..., T-1\}$, where worker $i = t \mod (N)$ do
2:      Compute gradient: $\boldsymbol{g}_t^{(i)}$ of worker $i$
3:      if $(t-1) \mod N = 0$ then
4:          Store the previous bias-corrected discrete second
            moment estimate: $\widehat{\boldsymbol{y}}_{prev} = \widehat{\boldsymbol{y}}$
```

```
5:      end if
6:      Compute the difference of the parameters:
        $\Delta_{t-N} = -(\boldsymbol{x}^{(i)}_{t-N+1} - \boldsymbol{x}_{prev})$
7:      Store the weights of worker $i$: $\boldsymbol{x}_{prev} = \boldsymbol{x}^{(i)}_{t-N+1}$
8:      $\boldsymbol{g}^{(i)}_t \leftarrow normalize\_gradient\left(\boldsymbol{g}^{(i)}_t\right)$
9:      Update second moment estimate:
        $\boldsymbol{y}_t = \beta_2 \boldsymbol{y}_{t-1} + [\boldsymbol{g}^{(i)}_t]^2$
10:     if $(t-1) \mod N = 0$ then
11:         Discretize bias-corrected second moment estimate:
            $\widehat{\boldsymbol{y}} = \boldsymbol{y}_t(1-\beta_2)/(1-\beta_2^t)$
12:     end if
13:     Update outer iterates of worker $i$:
        $\boldsymbol{x}^{(i)}_{t+1} = \boldsymbol{x}^{(i-1)}_t + STEP_t$
14: end for
```

Memory analysis

Procedure $gradient\_normalization$ uses two memory buffers. Algorithm 7.6 uses three memory buffers to store the previous value of the discretized second moment estimation in line 4, the value of the discrete second moment estimation in line 11, and the previous parameters of worker i in line 7. Therefore, Algorithm 7.6 uses five memory buffers for the entire algorithm and a buffer per each worker, resulting in $N+5$ buffers for N workers.

8 Experiments

We test MoRe on large scale vision and language tasks, using SGD with momentum and AdamW optimizers, respectively. We keep the canonical hyper-parameters used to train the networks, except for scaling of the minibatch sizes and the learning rate.

Table 1 summarizes the memory requirements of MoRe and DANA-GA as a function of the number of workers. For SGD with momentum, MoRe reduces memory requirements nearly by a factor of two, as compared to DANA-GA, while closely following the final test accuracy and convergence rate of the baseline, as shown in Table 2. For Adam, the memory requirements of MoRe and DANA-GA are almost the same. However, Table 3 shows that MoRe closely follows the convergence rate and the validation loss of the baseline while Table 4 shows that the convergence rate and final test accuracy of DANA-GA rapidly drop as the number of workers increase.

The most time-consuming operations are gradient computation and vector operations do not add a significant computational overhead. Still, Table 1 shows that MoRe and DANA-GA have a similar number of vector operations. Appendix G details the counting of vector operations.

Table 1 DANA-GA and MoRe computational overhead and memory requirements in the number of memory buffers for N workers, for SGD with momentum and Adam variants.

Optimizer	#Buffers	#Vector Ops
DANA-GA, SGD	2*N* + 3	23
MoRe, SGD	*N* + 3	20
DANA-GA, Adam	*N* + 4	28
MoRe, Adam	*N* + 5	30

While MoRe is designed for pipelined execution, our experiments take the cyclic-buffer simulation approach described in section 6.3, effectively reducing the problem into a DP ASGD equivalent with a predefined round-robin order of updates without requiring us to split the model for pipelined training. In turn, this enables us to isolate and benchmark the resulting convergence without introducing additional noise stemming from cumbersome model-splitting issues.

8.1 Experimental Setup

Since our experiments take the *DP ASGD approach with a round-robin order of updates*, we run them on a cluster of servers, each corresponding to a single asynchronous worker. On such a cluster, gradients can be simultaneously calculated on each server separately, and once calculated, a pipelined procedure can go over the gradients in a serial order and use them to update the worker models. However, for the sake of mathematical accuracy, the simultaneity of the execution is not important. Executing the algorithm on a real cluster with numerous nodes will clearly yield a significant wall-clock time speedup, but this makes the translation from MP to DP unnecessarily complex. It is much simpler to test the algorithm on a single server, iterating over the workers with each one computing its gradient from its current model. Such a procedure carefully maintains each worker's state outside the GPU memory and activates the state when the worker's turn arrives. Essentially, this is a more straightforward implementation of the cyclic-buffer approach described in section 6.3.

8.2 Total vs. Local Minibatch Size

For a certain number of workers in an experiment, as is the case for SSGD, we regard the total minibatch size as the per-worker local minibatch size multiplied by the number of workers, because all workers in ASGD compute gradients in parallel. Indeed, the only difference in this regard between ASGD and SSGD is that gradient computations at different workers are not synchronized in the former. For this reason, for a baseline with a minibatch of size M, we run the MoRe experiment on N workers with a local minibatch of size M/N.

8.3 Results

ImageNet

For image classification over the ImageNet ILSVRC2012 dataset [36], we use the ResNet-50 variant from NVIDIA's implementation [38] of ResNet v1.5 [37]. We run experiments on the total minibatch size of 2048 for 90 epochs, using Goyal's [30] canonical parameterization. In Appendix F.1, we provide training details.

In Table 2, we compare the final test accuracy and memory requirements of the baseline, MoRe, and DANA-GA. Since DANA-GA closely follows the baseline with SGD with momentum, we do not show its final test accuracy. The table shows that MoRe reduces the memory requirements almost by a factor of two, compared to DANA-GA, while closely following the final test accuracy of the baseline, as in DANA-GA. We provide more experimental results in Appendix C.1.

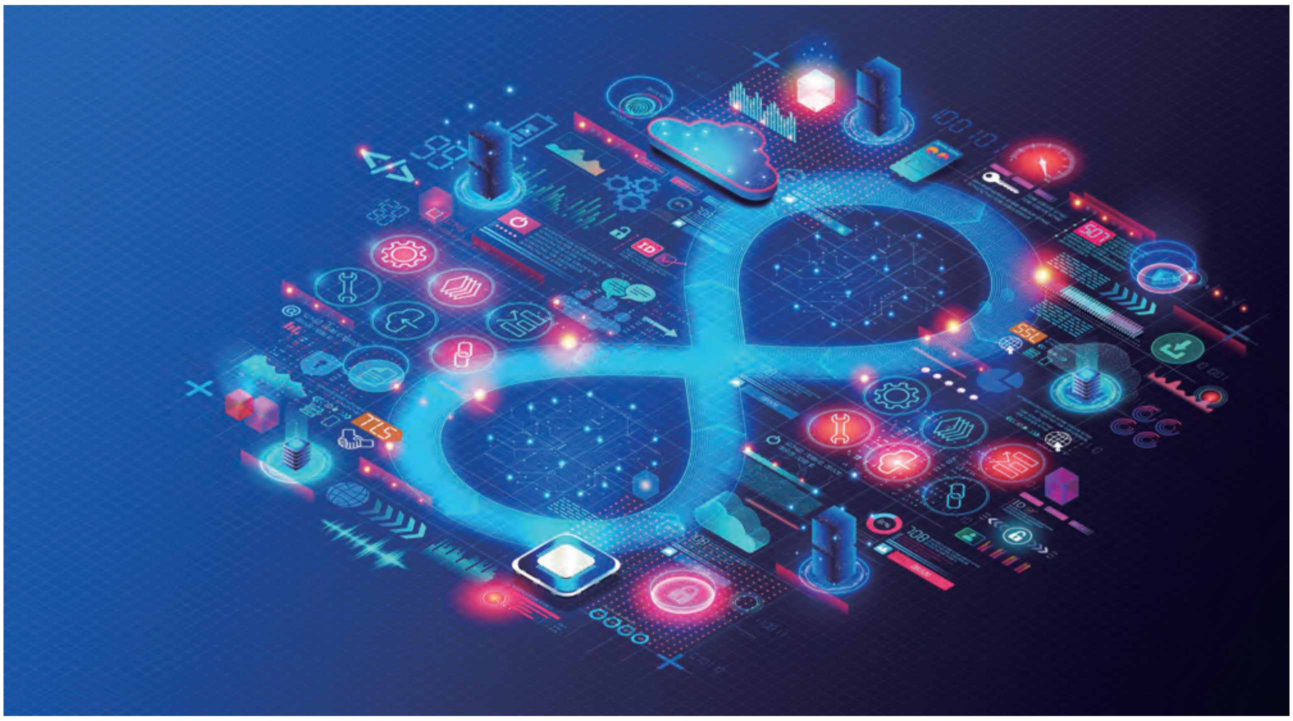
Table 2 Top-1 validation accuracy on ResNet50+ImageNet, SGD with momentum. Each row is based on five experiments. Baseline validation accuracy is 76.54%.

	MoRe			DANA-GA	
#Workers	Test Accuracy	STD	#Buffers	Test Accuracy	#Buffers
2w	76.37%	0.063	**5**	-	7
4w	76.29%	0.092	**7**	-	11
8w	76.28%	0.085	**11**	-	19

GPT-3

We base our implementation on NVIDIA's Megatron-LM GPT-3 [39] by replacing its internal optimizer step with MoRe's update rule. The dataset used for training is English Wikipedia [40]. We run two GPT variants — Small and Medium — and compare several scales, i.e., the number of MoRe workers against their respective baseline, keeping the total batch size constant at 512 sequences. The total number of samples used for training in all experiments is 19,200,000. We provide full training details in Appendix F.2.

In all scales, the validation loss of MoRe is comparable to its baseline, as can be seen from our experiments on GPT-3



Small and Medium in Table 3, where we summarize the final validation loss of MoRe and the memory requirements of MoRe and DANA-GA. We see that MoRe closely follows the baseline, while using nearly the same number of memory buffers as DANA-GA. We provide more experimental details in Appendix C.2.

Table 3 Final validation loss and the number of memory buffers of GPT-3 Small and Medium

	Loss GPT Small	Loss GPT Medium	#Buffers
Baseline	1.4732	1.356	-
MoRe 2w	1.4466	1.343	**7**
DANA-GA 2w	-	-	6
MoRe 4w	1.474	1.368	**9**
DANA-GA 4w	-	-	8
MoRe 8w	1.4872	1.384	**13**
DANA-GA 8w	-	-	12

Next, we point out that DANA-GA, Adam does not use the prediction of the master parameters like DANA-GA, SGD with momentum. For this reason, its test accuracy is expected to drop, compared to the baseline. Indeed, Table 4 shows the experimental results from [25, Table 2], which demonstrate that the final test perplexity rapidly falls as the number of workers increase. MoRe, on the other hand, uses almost the same amount of memory, while its validation loss closely follows the baseline.

Table 4 Final test perplexity of DANA-GA, Adam, using Transformer-XL on WikiText-103 and the degradation of the final test perplexity in %

	Test Perplexity	Degradation in %
Baseline	24.25	-
DANA-GA 4w	26.48	8.4%
DANA-GA 8w	28.7	15.5%

9 Conclusion

We propose MoRe — a new asynchronous training technique — and develop the family of asynchronous MP pipeline algorithms that achieve the convergence properties of SotA ASGD algorithms while reducing their memory requirements by a factor of two. This allows us to train larger models on the same hardware with a negligible computational overhead. We validate our findings via experiments over various large problems and show that for the ImageNet task, MoRe closely follows the convergence rate and the final test accuracy of the baseline, while reducing the memory requirements of DANA-GA almost by a factor of two. For the large Transformer architecture, MoRe and DANA-GA have similar memory requirements on the one hand, but on the other hand, MoRe closely follows the convergence rate and the validation loss of its baseline, while the convergence rate and the final test accuracy of DANA-GA rapidly degrade as the workers increase.

References

[1] Krizhevsky Alex, Sutskever Ilya, and Hinton Geoffrey E. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[2] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CVPR*, 2015.

[3] K He, X Zhang, S Ren, and J Sun. Deep residual learning for image recognition. *CVPR*, 2016.

[4] G. Botella and C. García. Real-time motion estimation for image and video processing applications. *Real-Time Image Process*., 2016.

[5] Z.-Q. Zhao, P. Zheng, S.-T. Xu, and X. Wu. Object detection with deep learning: A review. *IEEE Trans. Pattern Anal*., 2019.

[6] N. Sanil, P. A. N. Venkat, V. Rakesh, R. Mallapur, and M. R. Ahmed. Deep learning techniques for obstacle detection and avoidance in driverless cars. *AISP*, 2020.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NIPS*, 2017.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[9] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[11] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient largescale language model training on gpu clusters using megatron-lm. *Supercomputing 2021*, 2021.

[12] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[13] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. *CoRR*, 2015.

[14] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. Taming momentum in a distributed asynchronous environment. *ArXiv*, 2019.

[15] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[16] Yujun Lin, Song Han, Huizi Mao, YuWang, andWilliam J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[17] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*, pages 560–569. PMLR, 2018.

[18] Heyang Qin, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. Simigrad: Fine-grained adaptive batching for large scale training using gradient similarity measurement. *Advances in Neural Information Processing Systems*, 34, 2021.

[19] Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Olli Saarikivi, Tianju Xu, Vadim Eksarevskiy, Jaliya Ekanayake, and Emad Barsoum. Scaling distributed training with

adaptive summation. *Proceedings of Machine Learning and Systems*, 3, 2021.

[20] Jianyu Wang and Gauri Joshi. Cooperative sgd: A unified framework for the design and analysis of local-update sgd algorithms. *Journal of Machine Learning Research*, 22(213):1–50, 2021.

[21] Yunfei Teng, Wenbo Gao, Francois Chalus, Anna E Choromanska, Donald Goldfarb, and Adrian Weller. Leader stochastic gradient descent for distributed training of deep learning models. *Advances in Neural Information Processing Systems*, 32:9824–9834, 2019.

[22] Tomoya Murata and Taiji Suzuki. Bias-variance reduced local sgd for less heterogeneous federated learning. *ICML*, 2021.

[23] Lingjing Kong, Tao Lin, Anastasiia Koloskova, Martin Jaggi, and Sebastian Stich. Consensus control for decentralized deep learning. *ICML*, 2021.

[24] Ligeng Zhu, Hongzhou Lin, Yao Lu, Yujun Lin, and Han Song. Delayed gradient averaging: Tolerate the communication latency in federated learning. *NeurIPS*, 2021.

[25] Saar Barkai, Ido Hakimi, and Assaf Schuster. Gap-aware mitigation of gradient staleness. *ICLR*, 2020.

[26] Tao Lin, Sebastian Stich, Kumar Patel, and Martin Jaggi. Don't use large mini-batches, use local sgd. *ICLR*, 2020.

[27] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnntraining. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[28] Yang Bowen, Zhang Jian, Li Jonathan, Re Christopher, Aberger Christopher, and De Sa Christopher. Pipemare: Asynchronous pipeline parallel DNN training. *MLSys*, 2021.

[29] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhiming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. *ICML*, 2017.

[30] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[31] JianyuWang, Vinayak Tantia, Nicolas Ballas, and Michael Rabbat. Slowmo: Improving communication-efficient distributed sgd with slow momentum. *ICLR*, 2020.

[32] Michael Zhang, James Lucas, Jimmy Ba, and Geoffrey E. Hinton. Lookahead optimizer: k steps forward, 1 step back. *NeuIPS*, 2019.

[33] Sebastian U. Stich. Local sgd converges fast and communicates little. *ICLR*, 2019.

[34] Jimmy Ba and Diederik Kingma. Adam: A method for stochastic optimization. *ICLR*, 2015.

[35] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *ICLR*, 2019.

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, and Michael Bernstein. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015.

[37] NVIDIA. Resnet 1.5. https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet 50 v1 5 for pytorch, 2019.

[38] NVIDIA. Nvidia deep learning examples. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/ConvNets/image classification, 2021.

[39] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Neelakantan Arvind, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[40] wikipedia. English Wikipedia, the free encyclopedia, 2012. [Online; accessed 29-September-2012].

A SlowMo Algorithm

We present the original SlowMo framework in Algorithm A.1. For completeness, we reiterate its main details. Each worker maintains a local copy of the parameters, $\boldsymbol{x}_{t,k}^{(i)}$. Base optimizer buffers, such as momentum buffers, may be either reset, maintained or averaged across workers. Each such an alternative result in a separate instantiation of the SlowMo algorithm. Slow momentum buffer $\boldsymbol{u}_t$ is always synchronized across all workers. After the τ base optimizer steps, the workers compute average of their weights $\boldsymbol{x}_{t,k}^{(i)}$ in line 6, e.g., using AllReduce. Then, the workers perform slow momentum update of model weights by computing momentum $\boldsymbol{u}_{t+1}$ and concluding the iteration in line 8 with updating their parameters $\boldsymbol{x}_{t+1,0}^{(i)}$.

Algorithm A.1: SlowMo

Input: Base optimizer with learning rate γ_t; Inner loop steps τ; Slow learning rate α; Slow momentum factor β; Number of worker nodes m. Initial point $\boldsymbol{x}_{0,0}$ and initial slow momentum buffer $\boldsymbol{u}_0 = \mathbf{0}$.

```
1: for $t \in \{0, 1, ..., T-1\}$ at worker $i$ in parallel do
2:     Reset/Maintain/Average base optimizer buffers
3:     for $k \in \{0, 1, ..., \tau-1\}$ do
4:         Base optimizer step: $\boldsymbol{x}_{t,k+1}^{(i)} = \boldsymbol{x}_{t,k}^{(i)} - \gamma_t \boldsymbol{d}_{t,k}^{(i)}$
5:     end for
6:     Exact-Average: $\boldsymbol{x}_{t,\tau} = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_{t,\tau}^{(i)}$
7:     Update slow momentum: $\boldsymbol{u}_{t+1} = \beta \boldsymbol{u}_t + \frac{1}{\gamma_t}(\boldsymbol{x}_{t,0} - \boldsymbol{x}_{t,\tau})$
8:     Update outer iterates: $\boldsymbol{x}_{t+1,0} = \boldsymbol{x}_{t,0} - \alpha\gamma_t \boldsymbol{u}_{t+1}$
9: end for
```

B Proofs

Proof of Lemma 7.1:

Proof.

$$\begin{aligned}
\boldsymbol{x}_{t+1}^{(i)} - \boldsymbol{x}_t^{(i-1)} &\overset{(a)}{=} \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1} - \boldsymbol{x}_t + \alpha\gamma_{t-1}\boldsymbol{u}_t \\
&\overset{(b)}{=} \boldsymbol{x}_t - \alpha\gamma_t \boldsymbol{m}_t^{(i)} - \alpha(\gamma_{t-1} - \gamma_t\beta)\boldsymbol{u}_t \\
&- \alpha\gamma_t(\beta\boldsymbol{u}_t + \boldsymbol{m}_t^{(i)}) - \boldsymbol{x}_t + \alpha\gamma_{t-1}\boldsymbol{u}_t \\
&= -2\alpha\gamma_t \boldsymbol{m}_t^{(i)} ,
\end{aligned}$$

where (a) follows from line 9 of Algorithm 7.2, and (b) follows from lines 7 and 8.

Proof of Lemma 7.2.

Proof. We start by expanding $\boldsymbol{x}_{t+1}^{(i)}$ in line 9 of Algorithm 7.2:

$$\begin{aligned}
\boldsymbol{x}_{t+1}^{(i)} &= \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1} \\
&\overset{(a)}{=} \boldsymbol{x}_t - \alpha\gamma_t \boldsymbol{m}_t^{(i)} - \alpha(\gamma_{t-1} - \gamma_t\beta)\boldsymbol{u}_t \\
&\quad - \alpha\gamma_t(\beta\boldsymbol{u}_t + \boldsymbol{m}_t^{(i)}) \\
&= \boldsymbol{x}_t - \alpha\gamma_{t-1}\boldsymbol{u}_t - 2\alpha\gamma_t \boldsymbol{m}_t^{(i)} \overset{(b)}{=} \boldsymbol{x}_t^{(i-1)} - 2\alpha\gamma_t \boldsymbol{m}_t^{(i)} \\
&\overset{(c)}{=} \boldsymbol{x}_t^{(i-1)} - 2\alpha\gamma_t(\mu \boldsymbol{m}_{t-N}^{(i)} + \boldsymbol{g}_t^{(i)}) \\
&\overset{(d)}{=} \boldsymbol{x}_t^{(i-1)} - 2\alpha\gamma_t \\
&\quad \left(-\frac{\mu}{2\alpha\gamma_{t-N}}(\boldsymbol{x}_{t-N+1}^{(i)} - \boldsymbol{x}_{prev}) + \boldsymbol{g}_t^{(i)}\right) \\
&= \boldsymbol{x}_t^{(i-1)} + \mu\frac{\gamma_t}{\gamma_{t-N}}(\boldsymbol{x}_{t-N+1}^{(i)} - \boldsymbol{x}_{prev}) - 2\alpha\gamma_t \boldsymbol{g}_t^{(i)},
\end{aligned} \tag{6}$$

where (a) follows from lines 7 and 8, (b) from line 9, (c) from line 6, and (d) from Lemma 7.1.

Proof of Lemma 7.3.

Proof. We start by developing the difference of two consecutive workers' parameters:

$$\begin{aligned}
\boldsymbol{x}_{t+1}^{(i)} - \boldsymbol{x}_t^{(i-1)} &\overset{(a)}{=} \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1} - \boldsymbol{x}_t + \alpha\gamma_{t-1}\boldsymbol{u}_t \\
&\overset{(b)}{=} \boldsymbol{x}_t - \alpha\gamma_t \boldsymbol{r}_t - \alpha(\gamma_{t-1} - \gamma_t\beta)\boldsymbol{u}_t - \\
&\quad \alpha\gamma_t(\beta\boldsymbol{u}_t + \boldsymbol{r}_t) - \boldsymbol{x}_t + \alpha\gamma_{t-1}\boldsymbol{u}_t \\
&= -2\alpha\gamma_t \boldsymbol{r}_t \overset{(c)}{=} -2\alpha\gamma_t\left(\frac{1}{\bar{V}_t}\frac{\widehat{\boldsymbol{m}}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} + \lambda\boldsymbol{x}_t^{(i-1)}\right) \\
&\overset{(d)}{=} -2\alpha\gamma_t\left(\frac{1}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\boldsymbol{m}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} + \lambda\boldsymbol{x}_t^{(i-1)}\right) ,
\end{aligned}$$

where (a) follows from line 13 of Algorithm 7.5, (b) follows from lines 11 and 12, (c) follows from line 10, and (d) follows from line 6.

Proof of Lemma 7.4.

Proof. We start by expanding $\boldsymbol{x}_{t+1}^{(i)}$ in line 13 of Algorithm 7.5:

$$\begin{aligned}
\boldsymbol{x}_{t+1}^{(i)} &= \boldsymbol{x}_{t+1} - \alpha\gamma_t \boldsymbol{u}_{t+1} \\
&\overset{(a)}{=} \boldsymbol{x}_t - \alpha\gamma_t \boldsymbol{r}_t - \alpha(\gamma_{t-1} - \gamma_t\beta)\boldsymbol{u}_t - \alpha\gamma_t(\beta\boldsymbol{u}_t + \boldsymbol{r}_t) \\
&= \boldsymbol{x}_t - \alpha\gamma_{t-1}\boldsymbol{u}_t - 2\alpha\gamma_t \boldsymbol{r}_t \overset{(b)}{=} \boldsymbol{x}_t^{(i-1)} - 2\alpha\gamma_t \boldsymbol{r}_t \\
&\overset{(c)}{=} \boldsymbol{x}_t^{(i-1)} - 2\alpha\gamma_t\left(\frac{1}{\bar{V}_t}\frac{\widehat{\boldsymbol{m}}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} + \lambda\boldsymbol{x}_t^{(i-1)}\right) \\
&\overset{(d)}{=} \boldsymbol{x}_t^{(i-1)} - \frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\boldsymbol{m}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} - 2\alpha\gamma_t\lambda\boldsymbol{x}_t^{(i-1)} \\
&= \boldsymbol{x}_t^{(i-1)} - \frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\mu\boldsymbol{m}_{t-N}^{(i)} + \boldsymbol{g}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} \\
&- 2\alpha\gamma_t\lambda\boldsymbol{x}_t^{(i-1)} \\
&= \boldsymbol{x}_t^{(i-1)} - \frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\mu\boldsymbol{m}_{t-N}^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} - \\
&\quad \frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\boldsymbol{g}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}} + \epsilon} - 2\alpha\gamma_t\lambda\boldsymbol{x}_t^{(i-1)},
\end{aligned} \tag{7}$$

where (a) follows from lines 11 and 12, (b) from line 11, (c) from line 10, and (d) follows from line 6. Next, we develop the second summand in (7):

$$
\begin{aligned}
&\frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\mu\boldsymbol{m}_{t-N}^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}}+\epsilon}\\
&\overset{(a)}{=} -\mu\frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{1}{\sqrt{\widehat{\boldsymbol{y}}}+\epsilon}\bar{V}_{t-N}\\
&\frac{1-\mu^{\lceil (t-N)/N\rceil}}{1-\mu}\left(\sqrt{\widehat{\boldsymbol{y}}_{prev}}+\epsilon\right)\\
&\odot\left(-\frac{1}{2\alpha\gamma_{t-N}}\Delta_{t-N}+\lambda\boldsymbol{x}_{t-N}^{(i-1)}\right)\\
&= -\mu\frac{\bar{V}_{t-N}}{\bar{V}_t}\frac{1-\mu^{\lceil (t-N)/N\rceil}}{1-\mu^{\lceil t/N\rceil}}\frac{\sqrt{\widehat{\boldsymbol{y}}_{prev}}+\epsilon}{\sqrt{\widehat{\boldsymbol{y}}}+\epsilon}\odot\\
&\left(-\frac{\gamma_t}{\gamma_{t-N}}\Delta_{t-N}+2\alpha\gamma_t\lambda\boldsymbol{x}_{t-N}^{(i-1)}\right), \qquad (8)
\end{aligned}
$$

where (a) follows from Lemma 7.3. Now, we assign (8) into (7)

$$
\begin{aligned}
\boldsymbol{x}_{t+1}^{(i)} &= \boldsymbol{x}_t^{(i-1)} + \mu\frac{\bar{V}_{t-N}}{\bar{V}_t}\frac{1-\mu^{\lceil (t-N)/N\rceil}}{1-\mu^{\lceil t/N\rceil}}\frac{\sqrt{\widehat{\boldsymbol{y}}_{prev}}+\epsilon}{\sqrt{\widehat{\boldsymbol{y}}}+\epsilon}\\
&\odot\left(-\frac{\gamma_t}{\gamma_{t-N}}\Delta_{t-N}+2\alpha\gamma_t\lambda\boldsymbol{x}_{t-N}^{(i-1)}\right)\\
&-\frac{2\alpha\gamma_t}{\bar{V}_t}\frac{1-\mu}{1-\mu^{\lceil t/N\rceil}}\frac{\boldsymbol{g}_t^{(i)}}{\sqrt{\widehat{\boldsymbol{y}}}+\epsilon}-2\alpha\gamma_t\lambda\boldsymbol{x}_t^{(i-1)}
\end{aligned}
$$

C More Experimental Results

C.1 ImageNet Experimental Results

As shown in Figure 7, while the convergence path of MoRe is slower than the baseline, it closes the gap and reaches the same final top-1 accuracy.

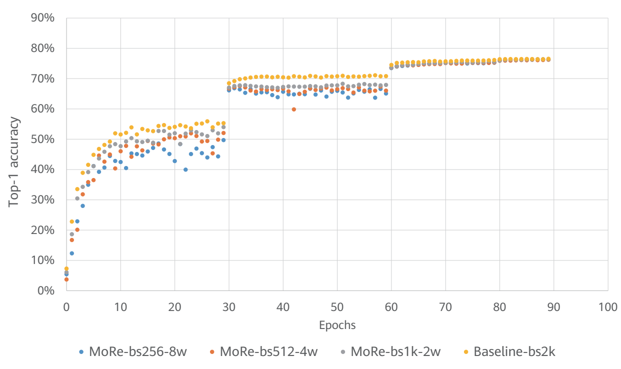


Figure 7 Top-1 validation accuracy on ResNet50+ImageNet

C.2 GPT Experimental Results

In all scales, it appears that the performance of MoRe is comparable to its baseline, as demonstrated by our experiments on GPT-3 Small and Medium in Figure 8 and Figure 9, respectively.

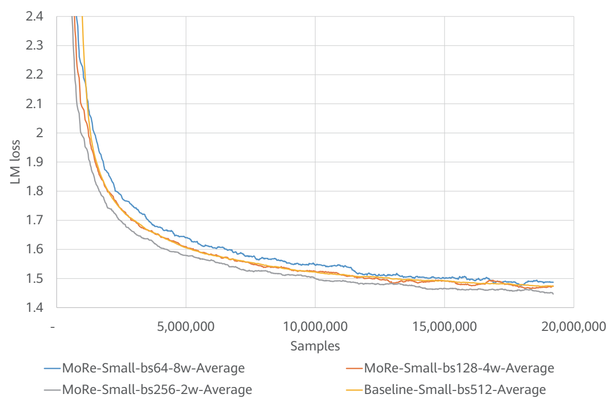


Figure 8 Validation loss of GPT-3 Small, mean value

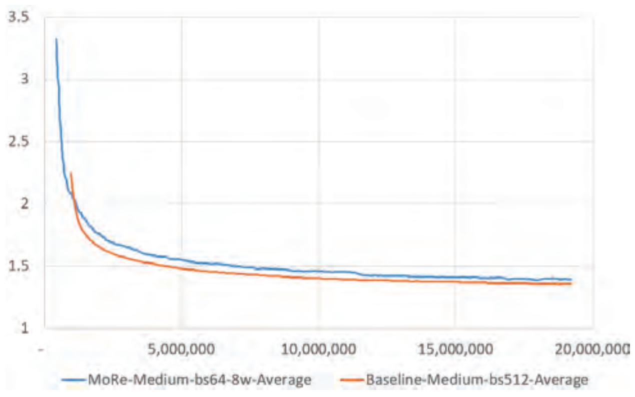


Figure 9 Validation loss of GPT-3 Medium, mean value

D Memory Requirements of the DANA-GA Algorithm

D.1 DANA-GA for SGD with Momentum

In this section, we analyze DANA-GA for SGD with momentum [25, Algorithm 8, DANA-Gap-Aware: master] to find out the number of memory buffers that it requires for running with N workers.

First, we present DANA-GA for SGD with momentum in Algorithms D.1, D.2 and D.3.

Now, we count the number of memory buffers in DANA-GA. One buffer is required to store the second moment estimate of C coefficient calculation, one is required to store the master weights, and one is required to manage the sum of the momentum values of all workers. Therefore, DANA-GA requires three buffers per entire algorithm. Now, each worker needs to store two buffers for weights and momentum. Therefore, for N workers, DANA-GA requires $2N+3$ buffers.

```
Algorithm D.1: DANA-Gap-Aware: worker
1: Always do:
2: Receive parameters $\theta_{k-\tau_k}$ from the master
3: Get $B$ training samples $\xi_{k-\tau_k}, [1\dots B]$
4: Compute gradient: $\boldsymbol{g}_k^i \leftarrow \sum_{b=1}^{B} \frac{\nabla F(\theta_{k-\tau_k};\xi_{k-\tau_k},b)}{B}$
5: Send $\boldsymbol{g}_k^i$ to the master
```

```
Algorithm D.2: DANA-Gap-Aware: master
1: Initialize the given weights for each worker: $\theta^i = \theta_0$
2: for $k = 1..K$ do
3:     Receive gradient $g_k^i$ from worker $i$
4:     Calculate Gap: $G_k = \frac{|\theta_k - \theta^i|}{C} + \mathbf{1}^d$
5:     Update worker's momentum $v^i \leftarrow \gamma v^i + \left(\frac{1}{G_k}\right) \odot g_k^i$
6:     Update master's weights $\theta_{k+1} \leftarrow \theta_k - \eta v^i$
7:     Save and send estimate $\theta^i \leftarrow \theta_{k+1} - \eta\gamma \sum_{j=1}^{N} v^j$ to worker $i$
8: end for
```

D.2 DANA-GA for Adam

In this section, we analyze DANA-GA for the Adam optimizer [25, Algorithm 11, Adam-Gap-Aware: master] to find the number of memory buffers it requires to run with N workers.

First, we present DANA-GA for Adam.

Now, we count the number of memory buffers in DANA-GA for Adam. One buffer is required to store the second moment estimate of C coefficient calculation, one is required to store the master weights, and two are required to store the first and second moment estimations. Therefore, DANA-GA requires four buffers per entire algorithm. Now, each worker needs to store one buffer for weights. Therefore, for N workers, DANA-GA requires $N+4$ buffers.

```
Algorithm D.3: $C$ coefficient calculation
Input: $\eta_{max}$ (usually $\eta_{max} = \eta_1$), $\beta_1 \in [0,1)$ exponential decay rates for the moment estimates
Initialize: $C \leftarrow \mathbf{0}^d$, $m_0 \leftarrow 0$
1: for $k = 1..K$ do
2:     Receive gradient $g_k^i$ from worker $i$
3:     Calculate update step: $v_{k+1} = \gamma v_k + g_k^i$
4:     Update biased second moment estimate
       $m_k \leftarrow \beta_1 m_{k-1} + (1-\beta_1) v_{k+1}^2$
5:     Compute bias-corrected second moment estimate
       $\hat{m}_k \leftarrow \frac{m_k}{1-\beta_1^k}$
6:     Calculate coefficient $C \leftarrow \eta_{max}\left(\sqrt{\hat{m}_k} + \epsilon\right)$
7: end for
```

```
Algorithm D.4: Adam-Gap-Aware: master
Require: $\eta_1 \dots \eta_K$: step lengths
Require: $\beta_1, \beta_2 \in [0,1)$: exponential decay rates for the moment estimates
Initialize: $m_0 \leftarrow 0,\ v_0 \leftarrow 0$
Initialize: $\theta^i = \theta_0$: parameters for every worker
1: for $k = 1..K$ do
2:     Receive gradient $g_k^i$ from worker $i$
3:     Calculate Gap: $G_k = \frac{|\theta_k - \theta^i|}{C} + \mathbf{1}^d$
4:     Update the biased first moment estimate
       $m_k \leftarrow \beta_1 m_{k-1} + \left(\frac{1-\beta_1}{G_k}\right) \odot \boldsymbol{g}_k^i$
5:     Update biased second moment estimate
       $v_k \leftarrow \beta_2 v_{k-1} + (1-\beta_2)(\boldsymbol{g}_k^i)^2$
6:     Compute the bias-corrected first moment estimate
       $\hat{m}_k \leftarrow \frac{m_k}{1-\beta_1^k}$
7:     Compute the bias-corrected secwond moment estimate $\hat{v}_k \leftarrow \frac{v_k}{1-\beta_2^k}$
8:     Update master's weights $\theta_{k+1} \leftarrow \theta_k - \frac{\eta_k \cdot \hat{m}_k}{\sqrt{\hat{v}_k}+\epsilon}$
9:     Send $\theta_{k+1}$ to worker $i$
10:    Save worker $i$'s given parameters $\theta^i \leftarrow \theta_{k+1}$
11: end for
```

E Bias Correction

First, we show how to adapt initialization bias correction from [34] to our momentum calculation. In our algorithms, we update momentum according to $\boldsymbol{m}_{t+1} = \mu \boldsymbol{m}_t + \boldsymbol{g}_{t+1}$, which may be written as a function of the previous gradients

$$\boldsymbol{m}_t = \sum_{n=1}^{t} \mu^{t-n} \boldsymbol{g}_n. \tag{9}$$

Taking expectations of the left- and right-hand sides of (9)

$$\begin{aligned}\mathbb{E}[\boldsymbol{m}_t] &= \sum_{n=1}^{t} \mu^{t-n}\,\mathbb{E}[\boldsymbol{g}_n] \overset{(a)}{=} \mathbb{E}[\boldsymbol{g}_t] \sum_{n=1}^{t} \mu^{t-n} + \xi \\ &= \mathbb{E}[\boldsymbol{g}_t] \sum_{n=0}^{t-1} \mu^n + \xi = \mathbb{E}[\boldsymbol{g}_t] \cdot \frac{1-\mu^t}{1-\mu} + \xi,\end{aligned} \tag{10}$$

where (a) holds because $\mathbb{E}[\boldsymbol{g}_n]$ is not the same for all n and $\xi = 0$ if $\mathbb{E}[\boldsymbol{g}_n]$ is stationary; otherwise ξ can be kept small since the exponential decay rate μ can and should be chosen such that the exponential moving average assigns small weights to gradients $\boldsymbol{g}_n$, whose indexes n are significantly smaller than t. Expression (10) leads to the initialization bias correction via the division of the momentum by $\frac{1-\mu^t}{1-\mu}$.

F Training Details

F.1 ImageNet

Specifically, the base learning rate is 0.8, decayed by a factor of 0.1 on epochs 30, 60, and 80, and warmed up linearly over the first five epochs. The weight decay is $1e-4$ and the SGD momentum factor is 0.9. In MoRe experiments, each worker's minibatch size and learning rate are set as the respective value in the baseline divided by the number of workers (i.e., linear scaling).

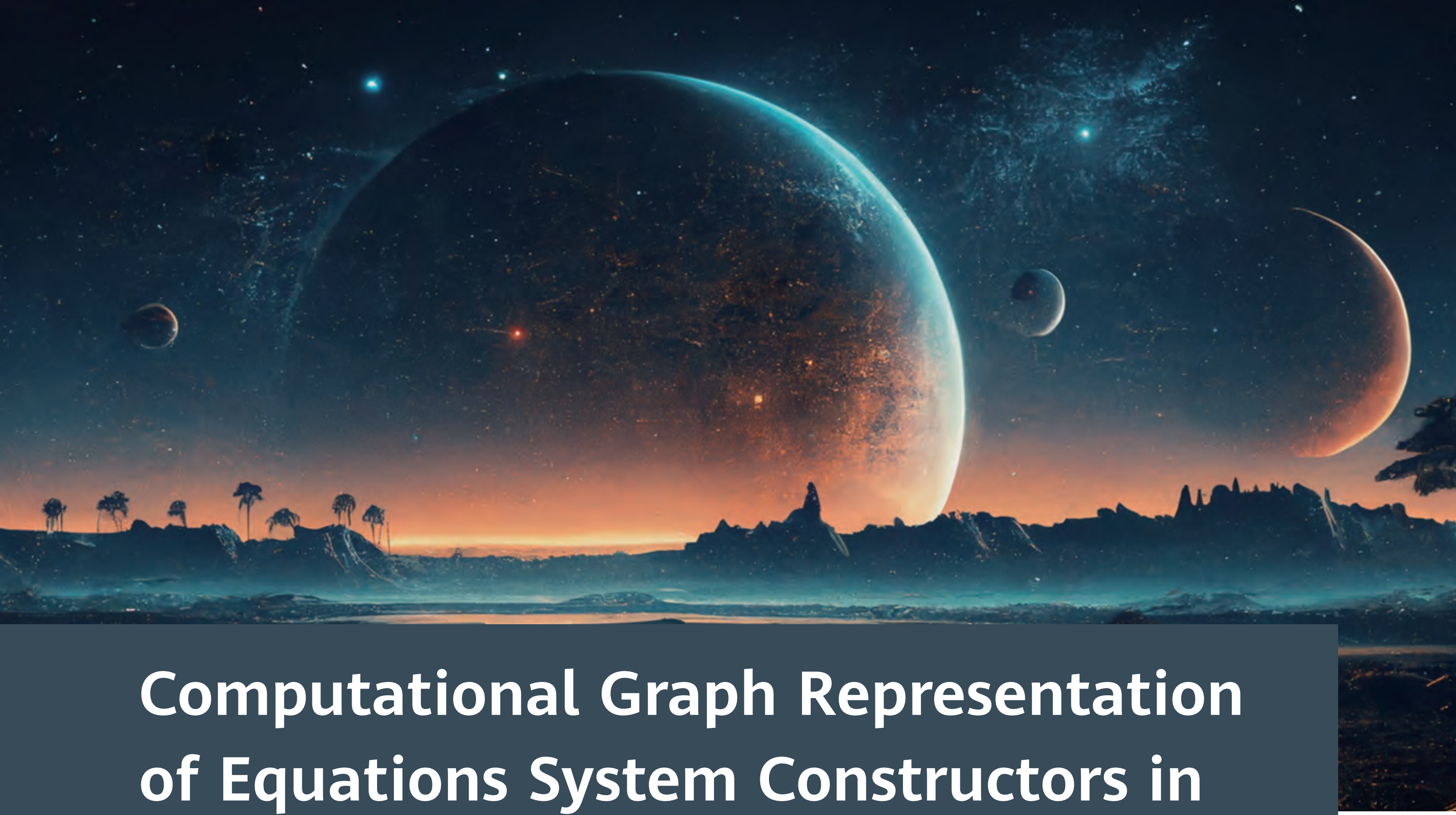
F.2 GPT-3

The base learning rate is $1.5e-4$, warmed up for 2% of the total number of iterations, and decayed with a cosine schedule. In MoRe experiments, each worker's minibatch size and learning rate are set as the respective value in the baseline divided by the number of workers (i.e., linear scaling). The total number of iterations is 37,500 in the baseline, as well as in MoRe, considering that an iteration in MoRe comprises a full round of per-worker iterations. Therefore, the total number of samples used for training in all experiments is 19,200,000.

G Counting Vector Operations

In this section, we detail the computational overhead of MoRe, compared to DANA-GA. To show this overhead, for each algorithm, we count the number of vector/vector and scalar/vector operations for a single iteration of the algorithm.

Note that in Algorithm 7.6, we can discretize the value $\widehat{\boldsymbol{y}} = \sqrt{\boldsymbol{y}_t(1-\beta_2)/(1-\beta_2^t)} + \epsilon$ in line 11. This discretization will lead to a simplified version of computing $STEP_t$, defined in Lemma 7.4. Therefore, we count the number of vector operations for Algorithm 7.6 assuming this simplification. The resulting counts are shown in Table 1.



Computational Graph Representation of Equations System Constructors in Hierarchical Circuit Simulation

Zichao Long, Lin Li, Lei Han, Xianglong Meng, Chongjun Ding, Ruiyan Li, Wu Jiang, Fuchen Ding, Jiaqing Yue, Zhichao Li, Yisheng Hu, Ding Li, Heng Liao *

Abstract

Equations system constructors of hierarchical circuits play a central role in device modeling, nonlinear equations solving, and circuit design automation. However, existing constructors present limitations in applications to different extents. For example, the costs of developing and reusing device models — especially coarse-grained equivalent models of circuit modules — remain high while parameter sensitivity analysis is complex and inefficient. Inspired by differentiable programming and leveraging the ecosystem benefits of open-source software, we propose an equations system constructor using the computational graph representation, along with its JSON format netlist, to address these limitations. This representation allows for runtime dependencies between signals and subcircuit/device parameters. The proposed method streamlines the model development process and facilitates end-to-end computation of gradients of equations remainders with respect to parameters. This paper discusses in detail the overarching concept of hierarchical subcircuit/device decomposition and nested invocation by drawing parallels to functions in programming languages, and introduces rules for parameters passing and gradient propagation across hierarchical circuit modules. The presented numerical examples, including (1) an uncoupled CMOS model representation using "equivalent circuit decomposition+dynamic parameters" and (2) operational amplifier (OpAmp) auto device sizing, have demonstrated that the proposed method supports circuit simulation and design and particularly subcircuit modeling with improved efficiency, simplicity, and decoupling compared to existing techniques.

Keywords

hierarchical circuit simulation, SPICE, computational graph, circuit static parameters, runtime variables, automatic differentiation, circuit behavior model, MOS sizing

* Corresponding author

1 Introduction

Analog circuit simulation (Figure 1) [1–3] has become one of the most important tools in the analog circuit EDA toolchain to assist verification and design. This is due to long-term research accumulation in algebraic differential equation theory [4–6], device modeling [7–9], equations system construction [10–12], nonlinear equations solvers [13, 14], hardware description languages (HDLs) [15–19], and more. However, we still see inconveniences with popular HDLs and simulators in reusing coarse-grained circuit module behavior models, introducing multi-physical effects, and applying gradient optimization methods to automatic design.

- For device and circuit modeling, HDL technology is often used to create a device behavior model, which is then compiled [17] into a program that the simulator can invoke. In particular, multi-port circuit behavior modeling often goes through two steps: function fitting and HDL implementation, for example, neural network fitting [20, 21]+HDL or Volterra polynomial [22]+SPICE netlists [23, 24].
- To achieve an optimal combination of circuit parameters in the automation of analog circuit design [25–27], it is necessary to first optimize the device size. Prior to 2010, researchers developed solutions based on gradient optimization using parameter sensitivity analysis [28, 29]. To tune design variables, modern process and software technologies require that software uses callbacks to modify model parameters. However, such an approach hinders users from obtaining gradient information of the variables. This has pushed recent research to shift its focus toward black-box methods, such as local sampling for gradient reconstruction [30–32], surrogate models [33–36], and reinforcement learning [37].

Many works attempt to address the above inconveniences from both the model compilation and the gradient acquisition perspectives. For example, Mahmutoglu *et al.* [38] developed the Verilog-AMS compiler that can run using Matlab/Octave, Kuthe *et al.* [39] can obtain more information about the equations and internal derivatives when the Verilog-AMS circuit module is compiled, and Hu *et al.* [40] provides an efficient implementation of adjoint equations for transient simulation. However, none of these works explore functional support possibilities from the perspective of systems equations construction.

Indeed, one significant cause of these inconveniences is that the analog HDLs contain many complex and even bloated features, which are necessary for these HDLs to simultaneously support the representation of structural and behavioral information. Such features include the automatic differentiation required for analog simulation, and polynomial interpolation that may be used in modeling [15, Sections 4.5.6, 9.21]. The intertwining of structural and behavioral information has isolated analog HDLs from the open ecosystems of other programming languages and tools, and also created high barriers for developing analog EDA tools. Furthermore, only static circuit parameters independent of signal values can be passed between nested circuit modules and simulation runtime variables are allowed only within modules ([15, Sections 3.4, 6],[16, Section 4.10]) — this is not conducive to reuse and development of coarse-grained circuit models.

Thriving technologies such as deep learning [41] and automatic differential programming [42] have contributed to a number of research fields in scientific computing, including datadriven multi-scale modeling and inverse problems [43–45]. Inspired by such works, this paper proposes a computational graph implementation of an equations system constructor that works in hierarchical

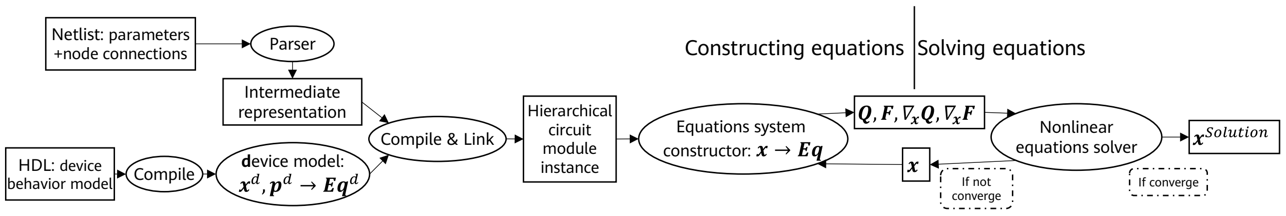

Netlist: parameters +node connections
Parser
Intermediate representation
HDL: device behavior model
Compile
device model: $x^d, p^d \rightarrow Eq^d$
Compile & Link
Hierarchical circuit module instance
Constructing equations
Solving equations
Equations system constructor: $x \rightarrow Eq$
$Q, F, \nabla_x Q, \nabla_x F$
Nonlinear equations solver
$x^{Solution}$
x
If not converge
If converge

Figure 1 Simulator flowchart

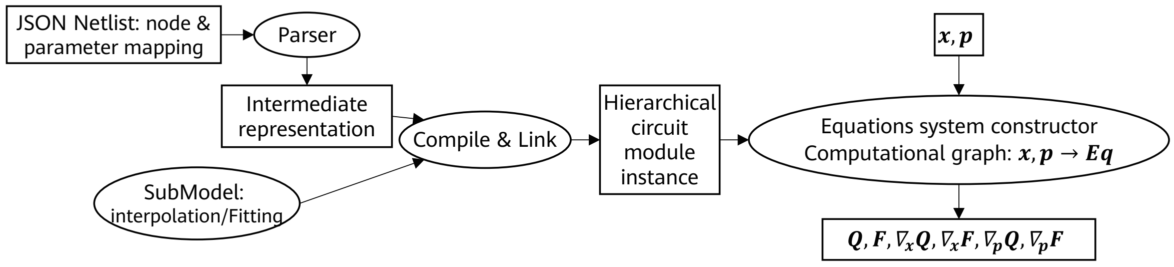

JSON Netlist: node & parameter mapping
Parser
Intermediate representation
SubModel: interpolation/Fitting
Compile & Link
Hierarchical circuit module instance
$\boldsymbol{x}, \boldsymbol{p}$
Equations system constructor
Computational graph: $\boldsymbol{x}, \boldsymbol{p} \to \boldsymbol{Eq}$
$\boldsymbol{Q}, \boldsymbol{F}, \nabla_x \boldsymbol{Q}, \nabla_x \boldsymbol{F}, \nabla_p \boldsymbol{Q}, \nabla_p \boldsymbol{F}$

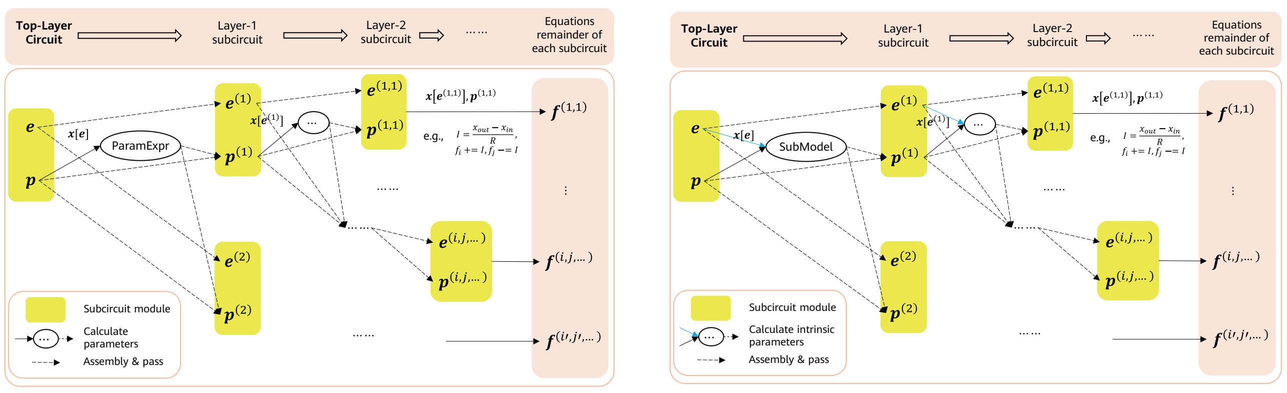
Figure 2 Flowchart: creating a computational graph from a JSON netlist

circuit simulation [12, 46–48], along with the corresponding JSON netlist and compilation method (Figure 2). The internal and external variables, design variables, model input parameters, and corresponding gradients of circuit modules are processed in a unified manner. This work takes circuit modules as the basic units of a computational graph and supports decoupled representation of circuit models using "equivalent circuit decomposition using JSON netlist+submodel for computing dynamic parameters". The advantages of this approach are twofold: (1) JSON format is easy to parse, and its basic data types "dictionary+list" are sufficient to represent circuit structure information. And (2), a submodel can be implemented with the help of the automatic differentiation capability of Julia[49]. Based on these two advantages, the proposed method in this paper simplifies circuit modeling and enhances the gradient acquisition capability of simulation tools.

Section 2 describes the processing of structural information related to circuit modules in a computational graph. Similar to defining, compiling, representing, and executing functions in a programming language [50–52], the process involves the following steps: module definition (netlist), parsing and compilation, data structures of module instances, and graph executor (Figure 3). Additionally, Section 2.4 introduces how to define and use SubModel for computing intrinsic dynamic parameters to provide behavioral information of each circuit module. Section 3 presents two application examples, one for device modeling and the other for a joint solution of DC/AC analysis and device sizing under different process, voltage, and temperature (PVT) combinations.

2 Hierarchical Circuit Equations System Constructor: Computational Graph

For any N generalized signals (i.e., equation unknowns) $\boldsymbol{x} \in \mathbb{R}^N$ and M signal-independent input variables or parameters $\boldsymbol{p} \in \mathbb{R}^M$, the mathematical meaning of circuit simulation is constructing and solving the following conservation algebraic differential equations [6, 14, 40]



(a) Existing method using static parameters

(b) Computational graph using dynamic parameters

Figure 3 Equations system constructor for hierarchical circuits: existing method v.s. computational graph. Each computing unit corresponds to a subcircuit, which can be decomposed into smaller subcircuits – the smallest granularity subcircuit are the "basic elements". $e^{(\cdots)}$ denotes the internal and external nodes of a subcircuit, $\boldsymbol{x}$ a generalized signal, such as node bias or branch current, $\boldsymbol{p}^{(\cdots)}$ the input parameters of a subcircuit, for example, the device size (alternatively, a non-linear capacitance, inductance, or current value of a basic element, functioning as a dynamic parameter), and $\boldsymbol{f}^{(\cdots)}$ the contribution of a subcircuit to the remainder of the simulation equations. Calculation of the Jacobian matrix is omitted in this schematic. In the existing method, a lower-layer parameter of a circuit module does not depend on the signal value ($\boldsymbol{x}[e]$), and calculation of ParamExpr can be completed when the netlist is built. In the computational graph method, the dynamic parameters are derived from the ntrinsic parameters output by SubModel – these parameters are obtained at the calculation runtime of the systems of equations.

$$\boldsymbol{f}(\dot{\boldsymbol{x}}(t), \boldsymbol{x}(t), \boldsymbol{p}) \triangleq \frac{\mathrm{d}\boldsymbol{Q}(\boldsymbol{x}, \boldsymbol{p})}{\mathrm{d}t} + \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{p}) = \boldsymbol{0}, \qquad \text{(Flat)}$$

where, $\boldsymbol{Q}$ denotes the dynamic part of the remainder of the equation (e.g., the charge of different capacitors and magnetic flux of different inductors) and $\boldsymbol{F}$ denotes the static part of the remainder of the equation (e.g., the total input DC current at each node and the voltage drop equation). Modern simulators tend to decompose and construct Equation (Flat) based on circuit hierarchy, which are easier to understand and parallelize:

$$\begin{aligned} \boldsymbol{f}(\dot{\boldsymbol{x}}, \boldsymbol{x}, \boldsymbol{p}) &= \boldsymbol{f}^{(1)}(\dot{\boldsymbol{x}}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{p}^{(1)}) + \boldsymbol{f}^{(2)}(\dot{\boldsymbol{x}}^{(2)}, \boldsymbol{x}^{(2)}, \boldsymbol{p}^{(2)}) + \cdots \\ &= \boldsymbol{f}^{(1,1)} + \boldsymbol{f}^{(1,2)} + \cdots + \boldsymbol{f}^{(2,1)} + \boldsymbol{f}^{(2,2)} + \cdots, \qquad \text{(Hierarchical)} \\ &\cdots \end{aligned}$$

where, $\boldsymbol{x}^{(i)}$, $\boldsymbol{p}^{(i)}$, and $\boldsymbol{f}^{(i)}$ respectively denote the input signals, parameters or variables, and contribution to the remainder of the equation of subcircuit i of the given circuit. The overlapping part between $\boldsymbol{x}^{(i)}$ and $\boldsymbol{f}^{(i)}$ depends on the common nodes of the subcircuits. As shown in Eq (Hierarchical), $\boldsymbol{f}^{(i)}$ may be further decomposed into $\boldsymbol{f}^{(i,1)}, \boldsymbol{f}^{(i,2)}, \cdots$ as required.

Note that Equation (Flat) represents only transient (TRAN) equations, and Equation (Flat) is usually discretized in the time direction during numerical solution. At each time step, the system of algebraic equations is solved using the Newton-Raphson method [12, Section 7.1]

$$\text{Solve } \boldsymbol{x}, \text{ Subject to } \frac{1}{\beta \Delta t}\boldsymbol{Q}(\boldsymbol{x}, \boldsymbol{p}) + F(\boldsymbol{x}, \boldsymbol{p}) + \boldsymbol{b} = \boldsymbol{0},$$

where $\boldsymbol{Q}, \boldsymbol{F}$, and sparse Jacobian matrix $\nabla_{\boldsymbol{x}}\boldsymbol{Q}, \nabla_{\boldsymbol{x}}\boldsymbol{F}$ need to be repeatedly calculated. For other types of simulation such as DC analysis and AC small signal analysis, Equation (Flat) must be converted (Appendix A). Because the processing of each analysis equation is similar, the following uses TRAN analysis and a simple JSON netlist subcircuit definition (Code 1) as an example to describe how to denote the calculation of $\boldsymbol{Q}^{(\cdots)}, \boldsymbol{F}^{(\cdots)}, \nabla_{\boldsymbol{x}^{(\cdots)} \text{ or } \boldsymbol{p}^{(\cdots)}}\boldsymbol{Q}^{(\cdots)}$, and $\nabla_{\boldsymbol{x}^{(\cdots)} \text{ or } \boldsymbol{p}^{(\cdots)}}\boldsymbol{F}^{(\cdots)}$ in hierarchical circuit simulation as the forward and backward pass of a computational graph (Figure 3b).

2.1 Subcircuit Module Definition in JSON Format Netlist

Similar to Verilog-AMS [15, Section 6], we define a circuit module that contains five parts of information (Table 1): (1) external nodes; (2) internal nodes; (3) input parameters; (4) decomposition of internal subcircuits; and (5) intrinsic parameters.

Code 1: User-defined subcircuit named SizeDepResistor in the netlist: a resistor whose resistance is sizedependent

```
"SizeDepResistor":{ # Define the subcircuit module.
  "ExternalNodes":["l","r"],
  "InputParams":["Rlength","Rwidth"],
  "InternalNodes":[],
  "SubModel":{
    "Expr":"[1e2*Rlength/Rwidth,]",
    "IntrinsicParams":["RValue"]
  },
  "Schematic":{
     # Instantiate each subcircuit or element in the
module.
    "instanceR":{
      "MasterName":"resistor",
      "ExternalNodes":{"left":"l","right":"r"},
      "InputParams":{"resistance":"RValue"}
    }
  }
}
```

The "Schematic" field represents internal subcircuit decomposition, which includes zero or more instantiation statements of subcircuits/devices. Each instantiation statement in "Schematic" is composed of (1) an instance name; (2) a class name (or master name); (3) external node connections; and (4) input parameter values. Code 1 provides an example, where the subcircuit decomposition part involves only one instance.

- "instanceR" indicates the instance name.
- "MasterName" indicates that the instance is of the

Table 1 Subcircuit module definition

Information	Content	Field	Required or Not
Structural information (dictionary+list)	List of external node names List of internal node names List of input parameter names Internal subcircuit decomposition	ExternalNodes InternalNodes InputParams Schematic	Required Required Required Required
Behavioral information (differentiable function)	Submodel for calculating intrinsic parameters	SubModel	Optional

"resistor" class. The class, or master, can be a subcircuit module or a type of built-in supported basic element.

- "ExternalNodes" indicates that the two external nodes "left" and "right" of the instance are respectively connected to ports "l" and "r" of the module, i.e., "SizeDepResistor" here. In general case, the nodes connected to each instance in "Schematic" must come from "ExternalNodes" and "InternalNodes" of the given module.
- "InputParams" indicates that the parameter of the instance is the internal variable "RValue" calculated by "SubModel". The parameters referenced by the instance in "Schematic" must come from: (1) global variables; (2) "InputParams" of the module; (3) "IntrinsicParams" under "SubModel" (if any) of the model.

For more information about SubModel and its functionality, see Section 2.4.

2.2 Representation of Subcircuit Module Instances in a Program

The subcircuit definition should be compiled into an appropriate hierarchical data structure (Figure 4) so that the equations system constructor can efficiently invoke subcircuit modules. A compiled subcircuit module contains two parts: common computation rules of subcircuits of the same master (Table 3) and instance private data (Table 2) There are a few points to note:

1 The external nodes of a subcircuit are from its upper-layer subcircuits. The top-layer circuit is a closed system without external nodes.

2 Subcircuit instances may share external nodes with one another, but the internal nodes of a subcircuit instance are exclusive to itself. When instantiating a subcircuit, ensure that its internal nodes do not conflict with each other.

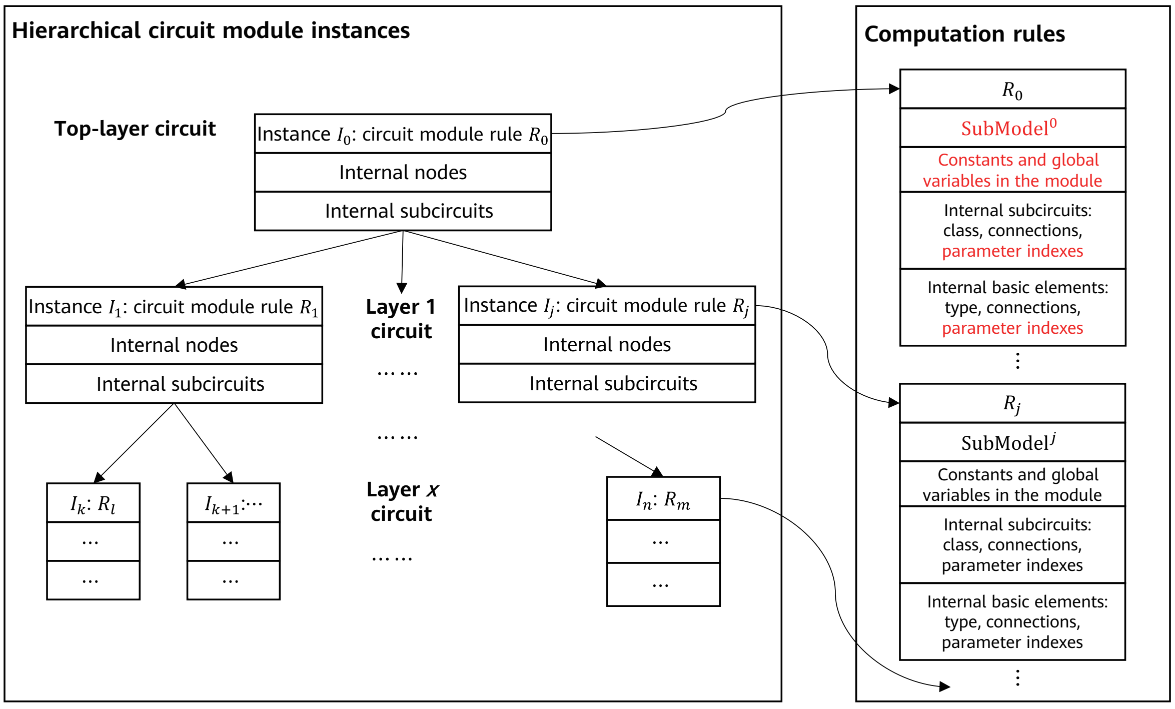

Hierarchical circuit module instances
Top-layer circuit
Instance I_0: circuit module rule R_0
Internal nodes
Internal subcircuits
Instance I_1: circuit module rule R_1
Internal nodes
Internal subcircuits
Layer 1 circuit
……
Instance I_j: circuit module rule R_j
Internal nodes
Internal subcircuits
……
I_k: R_l
I_{k+1}:⋯
Layer x circuit
……
I_n: R_m
Computation rules
R_0
SubModel^0
Constants and global variables in the module
Internal subcircuits: class, connections, parameter indexes
Internal basic elements: type, connections, parameter indexes
R_j
SubModel^j
Constants and global variables in the module
Internal subcircuits: class, connections, parameter indexes
Internal basic elements: type, connections, parameter indexes

Figure 4 Schematic diagram of hierarchical subcircuit module instances and computation rules. Text in red indicates the parts that mark the differences from the existing method [48]. Because the existing method does not need to support dynamic parameters, it is only necessary to store the fixed parameters of the devices in the computation rules of each circuit module. However, because the proposed method requires that parameters passed to lower-layer instances and devices be calculated during runtime, parameter indexing is necessary.

Table 2 Subcircuit module definition

Symbol	Description
rule	Pointer to the corresponding computation rules (Table 3)
in	Internal nodes
subckts	Pointers to lower-layer subcircuit instances

Table 3 Computation rule of a subcircuit module

Symbol	Description
c	Constants
gv	Global variables
SubModel	SubModel for calculating intrinsic parameters
SubCktsInfo	Lower-layer subcircuit nodes and parameter indexes
BasicElementInfo	Basic element nodes and parameter indexes

3 Global variables and system signals $\boldsymbol{x}$ are globally visible to all subcircuit modules. Only the indexes $\mathbf{gv}$ of global variables need to be stored in the module's computation rules (Table 3). The internal and external nodes of a module are also indexed for easy storage and passing.

4 If interactive analysis and debugging require more support information, the subcircuit master names, internal and external node parameter names, and lower-layer subcircuit instantiation statements can be added to a computation rule (Table 3). Additionally, the input parameters can be dynamically recorded in an instance (Table 2).

2.3 Instance Compilation from Subcircuit Module Definition

A JSON netlist file can be parsed using JSON parser tools available in a variety of programming languages. The compilation of the subcircuit module definition (Section 2.1) involves two steps:

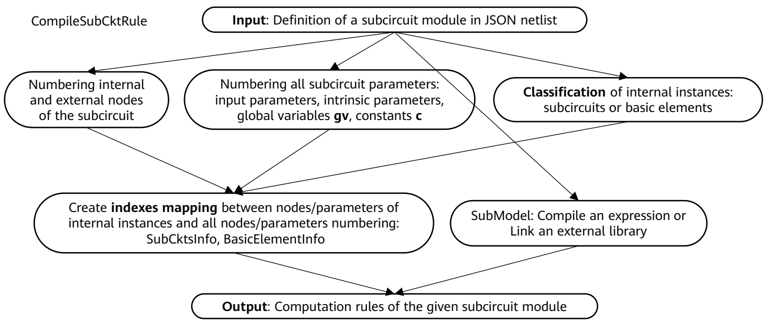
1 Compile the computation rules of all subcircuit modules (Figure 5a). SubModel finishes parsing and compilation based on the compiler's implementation. To process other structural information (i.e., to create nodes and parameters indexes), only basic algorithms and data structures such as lists and dictionaries are needed.

2 Recursively instantiate the hierarchical circuit modules (Figure 5b. The indexes of the input nodes are offset by $n = 0$ if the instantiation program was launched at top layer circuit. The proposed method ensures that the internal nodes of each subcircuit module are independent of each other.

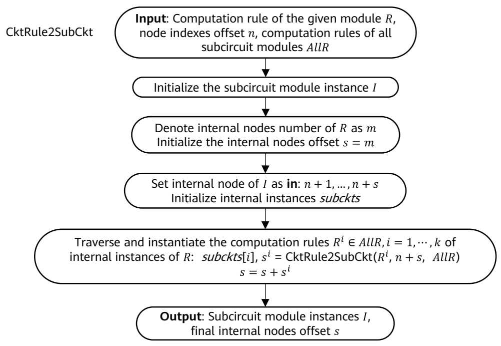
Note that the internal subcircuit decomposition of a circuit module may include both basic elements and other circuit modules defined in the netlist. As such, the compiler should be able to distinguish between these two types of instances before it creates indexes for nodes and parameters. This is in addition to the compiler being able to check that there are no circular definitions of subcircuit class, undefined subcircuit modules, disconnected subcircuits, and unused nodes in the circuit.

Basic Elements Basic elements are the smallest grained subcircuits without internal nodes or devices. Table 4 provides a brief list of supported basic elements. To add a type of basic elements, we need to define the electrical response function for each analysis and then provide information such as external nodes and input parameters to the compiler.

According to the modified nodal analysis method [11], the basic elements of the voltage source type must take the branch current as one of the degrees of freedom — this branch current is processed as an external GALV node in the compiler. At compile time, the GALV node needs to be


CompileSubCktRule
Input: Definition of a subcircuit module in JSON netlist
Numbering internal and external nodes of the subcircuit
Numbering all subcircuit parameters: input parameters, intrinsic parameters, global variables gv, constants c
Classification of internal instances: subcircuits or basic elements
Create indexes mapping between nodes/parameters of internal instances and all nodes/parameters numbering: SubCktsInfo, BasicElementInfo
SubModel: Compile an expression or Link an external library
Output: Computation rules of the given subcircuit module

(a) Compiling the computation rules of a module


CktRule2SubCkt
Input: Computation rule of the given module R, node indexes offset n, computation rules of all subcircuit modules $AllR$
Initialize the subcircuit module instance I
Denote internal nodes number of R as m
Initialize the internal nodes offset $s = m$
Set internal node of I as in: $n+1, \dots, n+s$
Initialize internal instances subckts
Traverse and instantiate the computation rules $R^i \in AllR, i = 1, \cdots, k$ of internal instances of R: $subckts[i], s^i$ = CktRule2SubCkt(R^i, $n+s$, $AllR$)
$s = s + s^i$
Output: Subcircuit module instances I, final internal nodes offset s

(b) Recursive instantiation

Figure 5 Compiling circuit modules

Table 4 Supported basic elements

MasterName	ExternalNodes	InputParams	Remarks
resistor	left,right	resistance	Resistor
capacitor	input,output	capacitance	Capacitor
inductor	input,output	inductance	Inductor
CS	input,output	current	Current Source
VS	input,output	voltage	Voltage Source
VCCS	left,right,input,output	MF	Voltage Controlled Current Source
CCCS	iorigin,input,output	MF	Current Controlled Current Source
VCVS	left,right,input,output	MF	Voltage Controlled Voltage Source
CCVS	iorigin,input,output	MF	Current Controlled Voltage Source

added to the upper-layer module as an internal node. A generalized external GALV node can also be added for basic elements of a non-voltage source type such as resistors, to indicate the current flowing through the element branch. Consider a TRAN analysis example, with the resistance of the resistor denoted as R, the left and right nodes l and r, the voltage value x_l and x_r, and with or without the GALV node and current value i, x_i, we can present the remainder of the equation corresponding to the resistor using a sparse vector as follows:

$$\textbf{Without external GALV: } \boldsymbol{Q}=\mathbf{0}, \boldsymbol{F}=[(l,-\tfrac{x_l-x_r}{R}),(r,\tfrac{x_l-x_r}{R})]$$

$$\textbf{With external GALV: } \boldsymbol{Q}=\mathbf{0}, \boldsymbol{F}=[(l,-x_i),(r,x_i),(i,x_r-x_l+R\cdot x_i)]$$

The two equations correspond to the so-called "element stamps" of the same type of elements described in [14, Section 2.4.4]. We may also consider them as two network analysis methods [10, 11] for the same type of elements, which requires the support of both the compiler and the equations system constructor. For different analysis type, the calculation of remainder terms and that of the gradients in the basic element simulation equation must be distinguished — we will not discuss that in detail here.

2.4 Execution: Forward and Backward Pass of a Computational Graph

Each basic computing unit of the computational graph (Figure 3b) corresponds to a subcircuit instance. When the subcircuit is invoked in a computational graph, the computing unit first takes external nodes and input variables as input from upper layer circuit. The compute unit then traverses the internal subcircuit and basic devices to calculate the equation remainder and the signal and variable gradients. Finally, these results are returned to the upper layer. See Algorithm 1 for the internal process details.

Figure 6 shows steps 1 to 5 of Algorithm 1, where **en**, **ip**, **in**, **gv**, and **intrp** stand for external nodes, input parameters, internal nodes, global variables, and intrinsic parameters, respectively. The internal and external nodes **en**, **in** of the circuit may be used to index the generalized signal values $\boldsymbol{x}[\mathbf{en}], \boldsymbol{x}[\mathbf{in}]$, respectively. The variables/parameters $\boldsymbol{p}$ involved in the circuit module consists of four parts: **ip**, **gv**, **intrp**, and **c**.

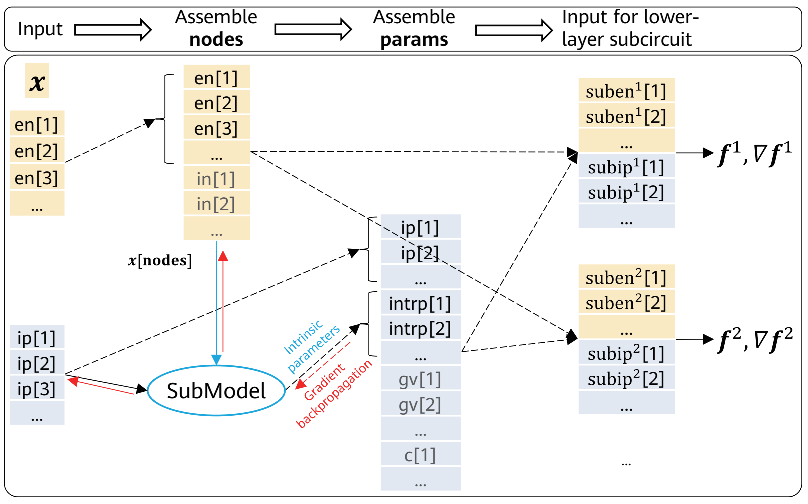


Figure 6 Schematic diagram of steps 1 to 5 of Algorithm 1, also a zoom-in view of the calls to the top-layer to layer-1 subcircuits in Figure 3b. In the existing method, the input and intrinsic parameters are computed at compile time, and no gradient of parameters are propagated backward at runtime.

```
Algorithm 1: Calling a Subcircuit
equations remainder, signal gradient, variable gradient =
EREvalCompositeSubCkt($\boldsymbol{x}$,ckt,en,ip)

Input: System signals $\boldsymbol{x}$, subcircuit instance ckt, external node indexes en, input parameters ip;
# Internal information of ckt: Internal nodes in, SubModel, global variables gv, constants c;
1. Assemble the internal nodes in and external nodes en of ckt, resulting in nodes=[en, in];
2. Calculate the intrinsic parameters according to the internal and external signals and input parameters: intrp = SubModel($\boldsymbol{x}$ [nodes], ip);
3. Assemble all variables and parameters of ckt, resulting in params= [ip, intrp, gv, c];
4. Extract the external nodes suben $\subset$ nodes and input parameters subip $\subset$ params of each subcircuit (subckt) in ckt from nodes, params} and call EvalCompositeSubCkt ($\boldsymbol{x}$, subckt, suben, subip);
5. Extract the external nodes and input parameters of each basic element in ckt from nodes, params, and calculate the equation remainder and gradient of each basic element;
6. Collect the remainder terms of all equations in steps 4 and 5;
7. Propagate signal and variable gradients of lower-layer subcircuits and basic elements backward according to the index mapping of steps 1 to 5;
Output: Equations remainder, signal gradient, variable gradient
```

SubModel and Intrinsic Parameters In the computational graph, SubModel takes the internal and external signals and input parameters of the module as inputs, and outputs all *intrinsic parameters* via **intrp**=SubModel(**signals**,**ip**), where **nodes**=[**en**,**in**], **signals**=$\boldsymbol{x}$[**nodes**], which can be passed to the lower-layer subcircuits and basic elements. This setup is based on the Assumption 1: "The behavior of a circuit module is and only is determined by the internal and external signals as well as input variables." Thus, for a SubModel, there is no need to perceive the internal signals of lower-layer subcircuits nor the node signals or parameters of other irrelevant modules. This setting can cover a considerably wide range of nonlinear effects and is easy to program.

Assumption 1. *The intrinsic parameters in a subcircuit module are uniquely determined by the bias signals of the internal and external nodes and the input parameters of the module.*

The submodel in the circuit definition should provide sufficient information so that the compiler can register the SubModel with the common rules (Table 3). In addition, there should be some protocol between the computational graph and the SubModel for obtaining the Jacobian matrix of **intrp** with respect to **signals**,**ip**

$$J_{\mathbf{s}} = \nabla_{\mathbf{signals}}\mathbf{intrp}, J_{\mathbf{ip}} = \nabla_{\mathbf{ip}}\mathbf{intrp}, \tag{1}$$

The specific implementation depends on the programming language used. As such, details are not provided here.

Layer-wise Gradient Backpropagation A computational graph completes the computation process by calling subcircuits. The computing logic differs from that involved in the existing method (Figure 3) in one major aspect: In the computational graph, the input parameters **subip** of lower-layer modules or elements come from a subset of the assembled parameters **params** (Figure 6). Consequently, gradient backpropagation for **subip** is required. The following describes the gradient backpropagation process in Algorithm 1 for TRAN simulation as an example.

For TRAN analysis, the returned value of Algorithm 1 contains the following eight items: $\boldsymbol{Q}, \boldsymbol{F}, \nabla_{\boldsymbol{x}}\boldsymbol{Q}, \nabla_{\boldsymbol{x}}\boldsymbol{F}$, $\nabla_{\mathbf{gv}}\boldsymbol{Q}, \nabla_{\mathbf{ip}}\boldsymbol{Q}$, $\nabla_{\mathbf{gv}}\boldsymbol{F}$, and $\nabla_{\mathbf{ip}}\boldsymbol{F}$. Because the gradient backpropagation of $\boldsymbol{Q}$ is the same as that of $\boldsymbol{F}$, only $\boldsymbol{Q}$ is considered for simplicity. The computation results of all subcircuits of Algorithm 1 are recorded as $\{\boldsymbol{Q}^i\}$, $\{\nabla_{\boldsymbol{x}}\boldsymbol{Q}^i\}$, $\{\nabla_{\mathbf{gv}}\boldsymbol{Q}^i\}$, and $\{\nabla_{\mathbf{subip}^i}\boldsymbol{Q}^i\}$, where the superscript i denotes the sequence number of the internal subcircuit or basic element. $\boldsymbol{Q}^i$, $\nabla_{\boldsymbol{x}}\boldsymbol{Q}^i$, and $\nabla_{\mathbf{gv}}\boldsymbol{Q}^i$ can be directly assembled as

$$\boldsymbol{Q} = \sum_i \boldsymbol{Q}^i, \nabla_{\boldsymbol{x}}\boldsymbol{Q} = \sum_i \nabla_{\boldsymbol{x}}\boldsymbol{Q}^i, \nabla_{\mathbf{gv}}\boldsymbol{Q} = \sum_i \nabla_{\mathbf{gv}}\boldsymbol{Q}^i,$$

while the gradient backpropagation of $\nabla_{\mathbf{subip}^i}\boldsymbol{Q}^i$ needs to be processed differently based on the index of $\mathbf{subip}^i$ to **params** = [**ip**, **intrp**, **gv**, **c**].

1 If $\mathrm{subip}^i[j] \in \mathbf{c}$, backpropagation is not performed.

2 If $\mathrm{subip}^i[j] \in \mathbf{ip} \cup \mathbf{gv}$, then $\nabla_{\mathrm{subip}^i[j]}\boldsymbol{Q}^i$ is propagated to the corresponding $\nabla_{\mathbf{ip}}\boldsymbol{Q}$ or $\nabla_{\mathbf{gv}}\boldsymbol{Q}$.

3 If $\mathrm{subip}^i[j] = \mathbf{intrp}[l]$ for any index l (Figure 6), given Assumption 1 and the Jacobian matrix of the intrinsic parameters with respect to the signal and input parameters (Equation 1), then (let $\boldsymbol{g} \triangleq \nabla_{\mathbf{subip}^i[j]}\boldsymbol{Q}^i$)

$$\nabla_{x[\mathbf{nodes}]}\boldsymbol{Q} \mathrel{+}= J_{\mathbf{s}}[:, l] \otimes \boldsymbol{g}, \nabla_{\mathbf{ip}}\boldsymbol{Q} \mathrel{+}= J_{\mathbf{ip}}[:, l] \otimes \boldsymbol{g}. \tag{2}$$

where, $\otimes$ represents the outer product of two vectors.

3 Applications

3.1 CMOS Device Model: Equivalent Circuit Decomposition + Dynamic Parameters

As mentioned earlier, any subcircuit module (e.g., CMOS) that satisfies Assumption 1 can be modeled as a submodel-based representation featuring "equivalent circuit decomposition + dynamic parameters". This section provides an implementation example based on a lookup table. Reimplementing BSIM model [7] as a SubModel is a conventional method that offers better compatibility with the existing method, but it is not adopted in this work. The CMOS module definition consists of the following elements (details about the definition are provided in Appendix B and the equivalent circuit diagram under AC analysis is given in Figure 7):

1 Internal and external nodes: **nodes** = [gate, source, drain, bulk];

2 Input parameters for the device size: **ip** = [MosL, MosW];

3 Intrinsic parameters output by SubModel: **intrp** = [ID, GDS, CDD, CSS, CGG, CGS, CGD, GM, GMB], whose value is determined by the four bias voltage values (**nodes**) and device size (ip).

The compiler loads external libraries and generates a function object of class "lut.MosLookup" to register it as a submodel in the subcircuit rule (Table 3). Among the intrinsic parameters, ID indicates the DC current between source and drain under DC analysis, and GDS, CDD, GM, etc. are equivalent small-signal parameters under AC analysis. There are a few points to note:

1 The role of built-in basic elements ICS and ACVCCS (Appendix B) is to ensure that ID only functions under DC analysis, while GDS, GM and GMB only function under AC analysis.

2 The DCAC hybrid analysis or DC analysis computational graph of the equation constructor can execute this circuit module. However, the pure AC analysis computational graph cannot independently run this circuit module: In order to establish the AC analysis equations, it is necessary to first compute [GDS,GM], etc., which are determined by the DC bias voltage. This is different from directly inducing small signal linear equations through TRAN analysis equations (Appendix A). In fact, Assumption 1 also stipulates that internal variables can depend on the bias voltage signal, but not on the small signals in linear analysis.

3 The SubModel can freely call external programs, such as using three-dimensional spline interpolation, provided that it ensures compliance with the interface requirements of the corresponding automatic differentiation system.

This submodel-based device model representation method features "equivalent circuit decomposition + dynamic parameters" and shows the following advantages:

1 Decoupled from circuit network analysis or simulation, the submodel is only responsible for calculating the intrinsic parameters and Jacobian matrix (Section 2.2). Circuit connectivity is not the submodel's concern.

2 The syntax and capability boundary of a submodel in calculating intrinsic parameters depend on the compiler's processing of the "SubModel" field in the netlist. This can be implemented easily using various external programs and automatic differential tools.

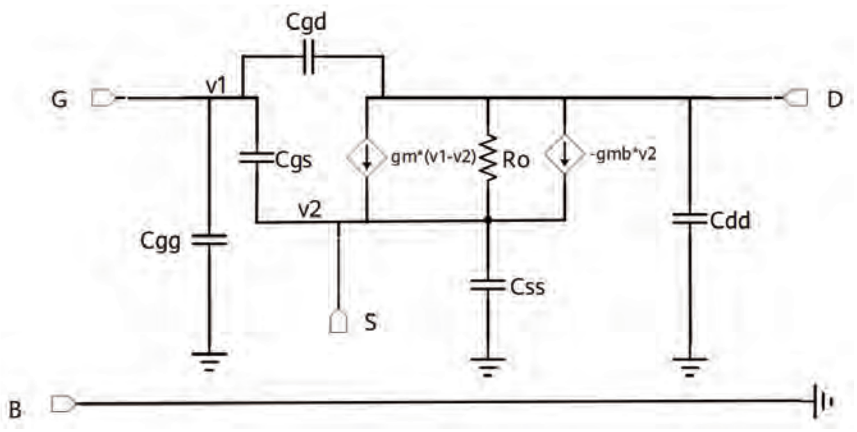


Figure 7 CMOS equivalent small-signal model [25, Figure 2.39], where, Ro is a resistor with resistance $\frac{1}{\mathrm{GDS}}$.

3.2 OpAmp Device Sizing: DC Operating Points Optimization Under Different PVT Combinations

Figure 8 shows the device sizing process in analog circuit design. Specifically, designers connect available devices accessible to the target process into circuits, and adjust the size of each device (such as a CMOS device) based on specific methodologies and experience so that a circuit can fulfill specification requirements under a given area and power consumption constraints. In this process, repeated circuit simulation is done to quantitatively inspect the behavior and performance of a circuit without the need to manufacture the physical circuit.

This process can be naturally converted into an optimization problem. The optimization strategy varies depending on whether the gradient is acquirable [28–37]. Take DC simulation as an example. To obtain the gradient, the DC steady-state equations $\boldsymbol{F}(\boldsymbol{x},\boldsymbol{p})=\mathbf{0}$ naturally provide an implicit mapping from the parameter $\boldsymbol{p}$ to the solution $\boldsymbol{x}^{solution}$ of the systems of equations, with the Jacobian matrix of this mapping being $\nabla_{\boldsymbol{p}}\boldsymbol{x}^{solution}=-\nabla_{\boldsymbol{x}}\boldsymbol{F}\backslash\nabla_{\boldsymbol{p}}\boldsymbol{F}$, where $\nabla_{\boldsymbol{x}}\boldsymbol{F},\nabla_{\boldsymbol{p}}\boldsymbol{F}$ may be directly given by the equations system construction method (computational graph 3b) described in Section 2. With this information, the gradient optimization method (Figure 9) can be used. Note that in an optimization process, the inverse of $\nabla_{\boldsymbol{x}}\boldsymbol{F}$ does not need to be completely solved. Instead, it is sufficient to solve a set of linear equations only once during each iteration's gradient backpropagation for a given loss function or constraint function l: $\nabla_{\boldsymbol{p}}l(\boldsymbol{x}^{solution})=(\nabla_{\boldsymbol{p}}\boldsymbol{x})^T\cdot\nabla_{\boldsymbol{x}}l=-(\nabla_{\boldsymbol{p}}\boldsymbol{F})^T\cdot\left(\nabla_{\boldsymbol{x}}\boldsymbol{F}^T\backslash\nabla_{\boldsymbol{x}}l\right)$

Taking an OpAmp (Figure 10a) as an example, consider the design variables of the circuit (i.e., the channel length and width of each MOSFET) as the variables to be optimized. Given the external current and voltage sources Ibias0, Ibias1, and V_{dd},V_+,V_- and the load resistance and capacitance $R_L=200\Omega, C_L=10^{-10}\mathrm{F}$, set the optimization goals as follows:

1 The DC operating points of all MOSFETs must be saturated under nine PVT conditions defined by $Corner\in[tt,ff,ss], Temperature\in[27,-40,125]$. For example, for an NMOS, its DC bias voltage must satisfy

$$\min(V_{gs},V_{ds},V_{sb},V_{gs}-V_{th})\geq 0,$$

where, V_{th} is subject to MosL,V_{gs},V_{ds}. For different PVT conditions, the SubModel needs to load different databases, and therefore the final simulation solutions obtained are also different.

2 Under the typical condition of $Corner=tt, Temperature=27$, slight fluctuation is allowed for voltage sources V_+,V_- as long as $V_++V_-=5v$ is satisfied. For the DC bias of V_{out}, the maximum must be greater than 4.35 V, and

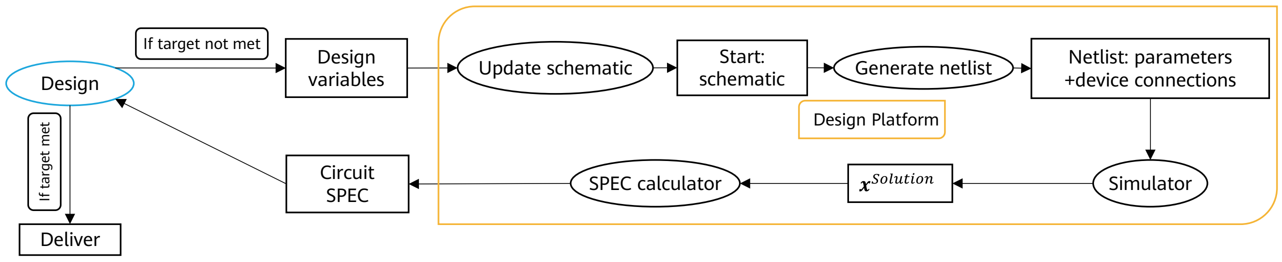


Figure 8 Manual device sizing: an iterative process

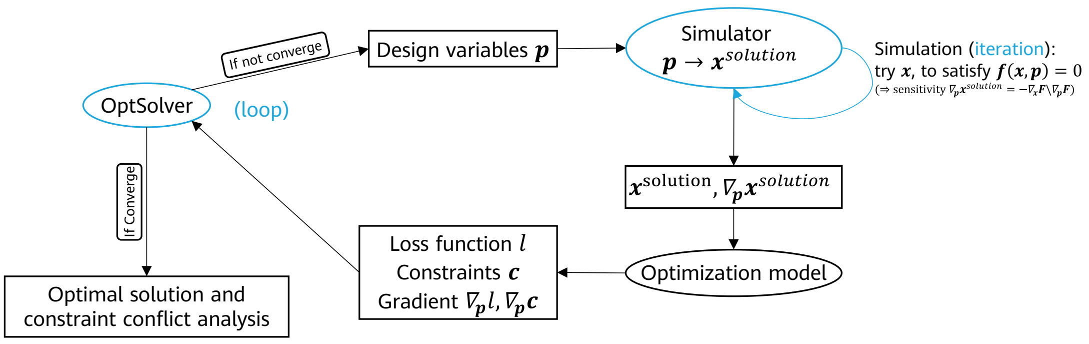
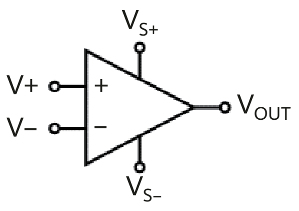
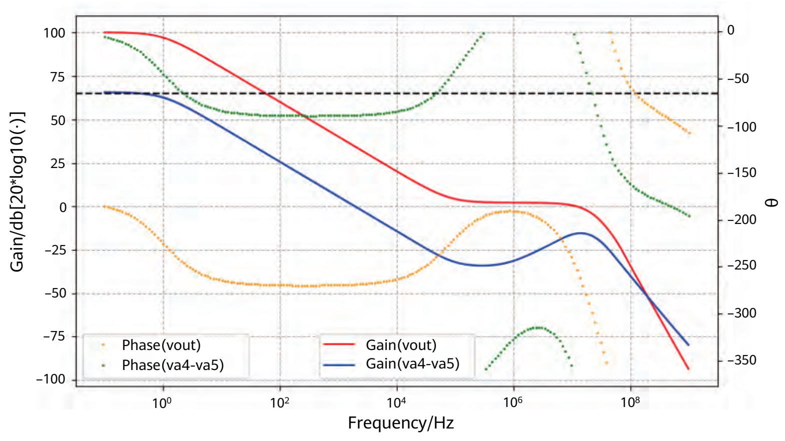

If not converge
Design variables p
Simulator
$p \to x^{solution}$
Simulation (iteration):
try x, to satisfy $f(x,p)=0$
(⇒ sensitivity $\nabla_p x^{solution} = -\nabla_x F \backslash \nabla_p F$)
OptSolver
(loop)
If Converge
$x^{solution}, \nabla_p x^{solution}$
Loss function l
Constraints c
Gradient $\nabla_p l, \nabla_p c$
Optimization model
Optimal solution and constraint conflict analysis

Figure 9 Automatic device sizing



(a) OpAmp schematic diagram [53]



(b) Input and output frequency response curves of the OpAmp after sizing

Figure 10 (a): OpAmp schematic diagram, including the bias circuit and the main circuit, and a total of 17 n-MOSFET and 17 p-MOSFET devices. At the DC operating point, when small-signal disturbance with a given frequency is applied at V_+, V_-, an output signal is detected at V_{out}. (**b**): Frequency response curves of OpAmp after sizing under $Corner = tt, Temperature = 27$ conditions, where, va4 and va5 are the two internal nodes in the circuit.

the minimum must be less than 0.3 V. Because our model (Section 3.1) is not used in the TRAN analysis, this requirement plays a similar role as the output swing indicator of circuits.

3 AC analysis is performed on the circuit under the typical condition $Corner = tt$, $Temperature = 27$ and a $v_{in+}, v_{in-} = \pm 0.5$ signal is applied at V_+, V_-. The DC gain $gain = 20 \cdot \log_{10}(|v_{out}|)$ of V_{out} must reach 100.

4 The design variables of each device meet given symmetry constraints. For example, the input MOSFET pair M_{n0_in}, M_{n0_ip} have the same size (MosL,MosW), and the current mirrors $M_{p30\_mirr}, M_{p20\_mirr}, M_{p10\_mirr},\cdot$ have the same channel length (MosL).

$$\begin{aligned}
&\min_{\boldsymbol{p}} l = \max(5 - \log_{10}(|\boldsymbol{v}[out]|), 0)^2 \\
\text{s.t.}\quad &\forall c \in [tt, ff, ss], t \in [27, -40, 125], \\
&\quad \boldsymbol{x}_L \preceq \boldsymbol{x}^{c,t} \preceq \boldsymbol{x}_U; Saturation(\boldsymbol{x}^{c,t}, \boldsymbol{p}) \succeq \boldsymbol{0}; \\
&\boldsymbol{x}^{down}[out] \le 0.3; \boldsymbol{x}^{up}[out] \ge 4.35; \\
&\boldsymbol{v} = \boldsymbol{A}^{tt,27} \backslash \boldsymbol{b}^{tt,27};\ C \cdot \boldsymbol{p} = \boldsymbol{0}.
\end{aligned} \tag{3}$$

This design task can be expressed as a constrained optimization problem (3). $\boldsymbol{p} \to \{\boldsymbol{x}^{c,t}\}, \boldsymbol{x}^{down}, \boldsymbol{x}^{up}$ is obtained by solving the system of DC equations under corresponding PVT conditions with input bias V_+, V_-. And $\boldsymbol{v} = \boldsymbol{A}^{tt,27} \backslash \boldsymbol{b}^{tt,27} \triangleq \boldsymbol{A} \backslash \boldsymbol{b}$ solves the system of AC linear equations under the $Corner = tt, temperature = 27$ condition, where the matrix elements of $\boldsymbol{A}$ are GM, GDS, etc. of each device subject to $\boldsymbol{x}^{tt,27}, \boldsymbol{p}$. We can use a computational graph of mixed DCAC analysis to calculate $\boldsymbol{A}, \boldsymbol{b}, \nabla_{\boldsymbol{x}} \boldsymbol{A}, \nabla_{\boldsymbol{x}} \boldsymbol{b}$ [1], which can further enable the calculation of $\nabla_{\boldsymbol{x}} l, \nabla_{\boldsymbol{p}} l$ (Appendix C). $C \cdot \boldsymbol{p} = \boldsymbol{0}$ represents the direct constraints on design variables, such as a symmetry constraint.

The optimization algorithm is implemented by using the open-source software Ipopt [54] and includes 72 variables, 27 equality constraints, and 308 inequality constraints to be solved. It took 356 seconds to run the whole process (including compiling Julia code and parsing netlist, etc.) on six threads on Intel(R) Core(TM) i7-8700 CPU at 3.20 GHz. Figure 10b shows the frequency response curve of the optimized circuit. The experimental results show that:

1 In hierarchical circuit simulation or sizing based on the computational graph, the device model and solution algorithm are decoupled from each other, allowing for high flexibility and efficiency.

2 The parameters in the computational graph are processed to function as variables, making gradient optimization of many indicators simpler and easier.

Note that the preceding experiments only consider the operation points of devices and circuit DC gains under typical conditions. To complete the design, more indicators (even discrete value indicators) need to be introduced into the optimization problem. There is no shortcut to properly integrating all indicators into the optimization framework, which however will not be addressed here.

4 Conclusion

In this paper, the static parameters of the circuit are processed as runtime variables in simulation, and the structural information and behavioral information of

[1] $A = i\omega \cdot \nabla_{\boldsymbol{x}} \boldsymbol{Q} + \nabla_{\boldsymbol{x}} \boldsymbol{F}$. For a simpler graph implementation, use the DCAC computational graph (instead of $\nabla \boldsymbol{Q}, \nabla \boldsymbol{F}$) to calculate ∇A. This avoids calculating and propagating backward the second derivative of $\boldsymbol{Q}, \boldsymbol{F}$ with regard to $\boldsymbol{x}$.

the circuit module/device are decoupled as "equivalent subcircuit decomposition + submodel-computed dynamic parameters". These further derive the computational graph representation of the equations system constructor for hierarchical circuits with circuit modules as the compute units of the computational graph. According to the two simple examples, this approach facilitates the decoupling and flexible interaction between netlists, models, and simulation and optimization algorithms. However, some problems exist with this approach. For example, because the variable gradient will be passed across the layers of a circuit, the topology analysis for circuit equations solvability and DAE-Index no longer works, requiring a more general hierarchical analysis theory. In the future, this approach will gain better generalization by supporting BSIM and more simulation types with more effects (i.e., S parameter or thermal effect) considered. Faster simulation is also possible if the program itself is optimized and the support for fast-SPICE technology is added.

Acknowledgment

I would like to thank my colleagues at Turing Department. It's been both a pleasure and an honor to work with them. This work has been also supported by many EDA and HDL experts, including Waisum Wang, Jiewen Fan, Zuochang Ye, Yang Lu, Shangxia Fang, Guoyong Shi, and Yu Ji. Special thanks go to Zhenya Zhou, Jiawei Zhuang, Long Wang, and Yuwei Fan for their kind help.

References

[1] Laurence Nagel and Ronald Rohrer. Computer analysis of nonlinear circuits, excluding radiation (CANCER). *IEEE Journal of Solid-State Circuits*, 6(4):166–182, 1971.

[2] WJ McCalla andWG Howard. BIAS-3-A program for the nonlinear dc analysis of bipolar transistor circuits. *IEEE Journal of Solid-State Circuits*, 6(1):14–19, 1971.

[3] Laurence W. Nagel and D.O. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html.

[4] Diana Estévez Schwarz and Caren Tischendorf. Structural analysis of electric circuits and consequences for MNA. *International Journal of Circuit Theory and Applications*, 28(2):131–162, 2000.

[5] Peter Kunkel and Volker Mehrmann. *Differential-algebraic equations: analysis and numerical solution*, volume 2. European Mathematical Society, 2006.

[6] Michael Günther, Uwe Feldmann, and Jan ter Maten. Modelling and discretization of circuit problems. *Handbook of numerical analysis*, 13:523–659, 2005.

[7] Yogesh Singh Chauhan, Sriram Venugopalan, Mohammed A Karim, Sourabh Khandelwal, Navid Paydavosi, Pankaj Thakur, Ali M Niknejad, and Chenming C Hu. BSIM—Industry standard compact MOSFET models. In *2012 Proceedings of the European Solid-State Device Research Conference* (ESSDERC), pages 46–49. IEEE, 2012.

[8] Tatsuya Ezaki, Hans Jurgen Mattausch, and Mitiko Miura-mattausch. *Physics And Modeling Of Mosfets, The: Surface-potential Model Hisim*. World Scientific, 2008.

[9] Gennady Gildenblat, Xin Li, Weimin Wu, Hailing Wang, Amit Jha, Ronald Van Langevelde, Geert DJ Smit, Andries J Scholten, and Dirk BM Klaassen. PSP: An advanced surface-potential-based MOSFET

model for circuit simulation. *IEEE Transactions on Electron Devices*, 53(9):1979–1993, 2006.

[10] Gary Hachtel, R Brayton, and Fred Gustavson. The sparse tableau approach to network analysis and design. *IEEE Transactions on circuit theory*, 18(1):101–113, 1971.

[11] Chung-Wen Ho, Albert Ruehli, and Pierce Brennan. The modified nodal approach to network analysis. *IEEE Transactions on circuits and systems*, 22(6):504–509, 1975.

[12] JG Fijnvandraat, SHMJ Houben, EJW ter Maten, and JMF Peters. Time domain analog circuit simulation. In *International Workshop on Computational codes: Technological Aspects of Mathematics*, 2002.

[13] Ognen Nastov, Rircardo Telichevesky, Ken Kundert, and Jacob White. Fundamentals of fast simulation algorithms for RF circuits. *Proceedings of the IEEE*, 95(3):600–621, 2007.

[14] Farid N Najm. *Circuit simulation*. John Wiley & Sons, 2010.

[15] HDL Verilog. Verilog-AMS Language Reference Manual. *Eda-Stds. Org,* 2014.

[16] IEEE Standards Association *et al.* IEEE standard for Verilog hardware description language (IEEE 1364-2005). *http://standards.ieee.org/*, 2006.

[17] Laurant Lemaitre, Colin McAndrew, and Steve Hamm. ADMS-automatic device model synthesizer. In *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference (Cat. No. 02CH37285)*, pages 27–30. IEEE, 2002.

[18] Ernst Christen and Kenneth Bakalar. VHDL-AMS-a hardware description language for analog and mixed-signal applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

[19] François Pêcheux, Christophe Lallement, and Alain Vachoux. VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems. *IEEE transactions on Computer-Aided design of integrated Circuits and Systems*, 24(2):204–225, 2005.

[20] Peter BL Meijer. Neural networks for device and circuit modelling. In *Scientific Computing in Electrical Engineering*, pages 251–258. Springer, 2001.

[21] Lining Zhang and Mansun Chan. Artificial neural network design for compact modeling of generic transistors. *Journal of Computational Electronics*, 16(3):825–832, 2017.

[22] Moning Zhang, Yang Tang, and Zuochang Ye. Large-signal MOSFET modeling using frequency-domain nonlinear system identification. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 626–632. IEEE, 2014.

[23] Zhe Zhang, Runsheng Wang, Cheng Chen, Qianqian Huang, Yangyuan Wang, Cheng Hu, Dehuang Wu, Joddy Wang, and Ru Huang. New-generation design-technology cooptimization (DTCO): Machine-learning assisted modeling framework. In 2019 *Silicon Nanoelectronics Workshop (SNW)*, pages 1–2. IEEE, 2019.

[24] Sitansusekhar Roymohapatra, Ganesh R Gore, Akanksha Yadav, Mahesh B Patil, Krishnan S Rengarajan, Subramanian S Iyer, and Maryam Shojaei Baghini. A novel hierarchical circuit LUT model for SOI technology for rapid prototyping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(5):1073–1083, 2019.

[25] Behzad Razavi. *Design of analog CMOS integrated circuits*. Tata McGraw-Hill Education, 2002.

[26] Flandre Silveira, Denis Flandre, and Paul GA Jespers. A g/sub m//I/sub D/based methodology for the design of CMOS analog circuits and its application to the synthesis of a silicon-on-insulator micropower OTA. *IEEE journal of solid-state circuits*, 31(9): 1314–1319, 1996.

[27] Paul Jespers. *The gm/ID Methodology, a sizing tool for low-voltage analog CMOS Circuits: The semi-empirical and compact model approaches*. Springer Science & Business Media, 2009.

[28] Yong Zhan and Sachin S Sapatnekar. Optimization of integrated spiral inductors using sequential quadratic programming. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 1, pages 622–627. IEEE, 2004.

[29] Bhavna Agrawal, Frank Liu, and Sani Nassif. Circuit optimization using scale based sensitivities. In *IEEE Custom Integrated Circuits Conference 2006,* pages 635–638. IEEE, 2006.

[30] Guanming Huang, Liuxi Qian, Siwat Saibua, Dian Zhou, and Xuan Zeng. An efficient optimization based method to evaluate the DRV of SRAM cells. *IEEE Transactions on Circuits and Systems I: Regular Papers,* 60(6):1511–1520, 2013.

[31] Arthur Nieuwoudt and Yehia Massoud. Multi-level approach for integrated spiral inductor optimization. In *Proceedings of the 42nd annual Design Automation Conference*, pages 648–651, 2005.

[32] Bo Peng, Fan Yang, Changhao Yan, Xuan Zeng, and Dian Zhou. Efficient multiple starting point optimization for automated analog circuit optimization via recycling simulation data. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1417–1422. IEEE, 2016.

[33] Alessandro Girardi and Lucas C Severo. Analog CMOS Design Automation Methodologies for Low-Power Applications. *Advances in Analog Circuits*, page 1, 2011.

[34] Wenlong Lyu, Fan Yang, Changhao Yan, Dian Zhou, and Xuan Zeng. Batch Bayesian optimization via multi-objective acquisition ensemble for automated analog circuit design. *In International conference on machine learning*, pages 3306–3314. PMLR, 2018.

[35] Ye Wang, Michael Orshansky, and Constantine Caramanis. Enabling efficient analog synthesis by coupling sparse regression and polynomial optimization. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[36] Wenlong Lyu, Pan Xue, Fan Yang, Changhao Yan, Zhiliang Hong, Xuan Zeng, and Dian Zhou. An efficient bayesian optimization approach for automated optimization of analog circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(6):1954–1967, 2017.

[37] Changcheng Tang, Zuochang Ye, and Yan Wang. Parametric circuit optimization with reinforcement learning. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 197–202. IEEE, 2018.

[38] A Mahmutoglu, Xufeng Wang, and Jaijeet Roychowdhury. New generation verilog-A model development tools: VAPP and VALint. *New Generation Verilog-A Model Development Tools: VAPP and VALint, Tech. Rep*, 2018.

[39] Pascal Kuthe, Markus Müller, and Michael Schröter. VerilogAE: An Open Source Verilog-A Compiler for Compact Model Parameter Extraction. *IEEE Journal of the Electron Devices Society*, 8:1416–1423, 2020.

[40] Wenfei Hu, Zuochang Ye, and Yan Wang. Adjoint transient sensitivity analysis for objective functions associated to many time points. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[41] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[42] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18:1–43, 2018.

[43] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, and EJPRL Weinan. Deep potential molecular dynamics: a scalable model with the accuracy of quantum mechanics. *Physical review letters*, 120(14):143001, 2018.

[44] Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. PDE-Net: Learning PDEs from data. In *International Conference on Machine Learning*, pages 3208–3216. PMLR, 2018.

[45] Zichao Long, Yiping Lu, and Bin Dong. PDE-Net 2.0: Learning PDEs from data with a numeric-symbolic hybrid deep network. *Journal of Computational Physics*, 399:108925, 2019.

[46] EJW Ter Maten. Numerical methods for frequency domain analysis of electronic circuits. *Survey on Mathematics for Industry*, 8:171–185, 1999.

[47] Tamal Mukherjee and Gary K Fedder. Hierarchical mixed-domain circuit simulation, synthesis and extraction methodology for MEMS. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(3):233–249, 1999.

[48] Andrei Tcherniaev, Iouri Feinberg, Walter Chan, Jeh-Fu Tuan, and An-Chang Deng. Transistor level circuit simulator using hierarchical data, June 10 2003. US Patent 6,577,992.

[49] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017. doi: 10.1137/ 141000671.

[50] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[51] Steven Muchnick *et al. Advanced compiler design implementation*. Morgan kaufmann, 1997.

[52] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[53] Wikipedia: Operational Amplifier.

[54] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.

Appendix

A TRAN Analysis Induced AC Analysis Equation

To solve the transient equation (Flat) and the DC steady-state equation $\boldsymbol{F}(\boldsymbol{x},\boldsymbol{p})=\boldsymbol{0}$, the Newton-Raphson method calculates $\boldsymbol{Q},\boldsymbol{F}\in\mathbb{R}^N$ and the Jacobian matrix $\nabla_{\boldsymbol{x}}\boldsymbol{Q},\nabla_{\boldsymbol{x}}\boldsymbol{F}\in\mathbb{R}^{N\times N}$ repeatedly. In AC small-signal analysis, consider applying perturbation $\delta\boldsymbol{x},\delta\boldsymbol{p}$ around the steady-state solution $\boldsymbol{x},\boldsymbol{p}$ and linearizing Equation (Flat) as follows:

$$\nabla_{\boldsymbol{x}}\boldsymbol{Q}\cdot\dot{\delta\boldsymbol{x}}+\nabla_{\boldsymbol{p}}\boldsymbol{Q}\cdot\dot{\delta\boldsymbol{p}}+\nabla_{\boldsymbol{x}}\boldsymbol{F}\cdot\delta\boldsymbol{x}+\nabla_{\boldsymbol{p}}\boldsymbol{F}\cdot\delta\boldsymbol{p}=\boldsymbol{0}$$

Let $\delta\boldsymbol{x},\delta\boldsymbol{p}$ be a small signal with an angular frequency of ω: $\delta\boldsymbol{x}=\boldsymbol{\epsilon_x}\cdot e^{i\omega t},\delta\boldsymbol{p}=\boldsymbol{\epsilon_p}\cdot e^{i\omega t}$. Then the system of linear equations of AC small-signal analysis is

$$(i\omega\cdot\nabla_{\boldsymbol{x}}\boldsymbol{Q}+\nabla_{\boldsymbol{x}}\boldsymbol{F})\cdot\boldsymbol{\epsilon_x}=(i\omega\cdot\nabla_{\boldsymbol{p}}\boldsymbol{Q}+\nabla_{\boldsymbol{p}}\boldsymbol{F})\cdot\boldsymbol{\epsilon_p}\tag{4}$$

B CMOS Subcircuit

Code 2: CMOS subcircuit

```
"NMOSTYPE":{
  "ExternalNodes":["gate","source","drain","bulk"],
  "InputParams":["MosL","MosW"],
  "InternalNodes":[],
  "SubModel":{
    "Analysis":["DC","TRAN"],
    "ModelLoader":"SimInfo->lut.MosLookup(\"NMOSTYPE\",
/path/to/data; SimInfo=SimInfo)",
    "IntrinsicParams":

["ID","GDS","CDD","CSS","CGG","CGS","CGD","GM","GMB"]
  },
  "Schematic":{
    "ids":{
      "MasterName":"ICS",

"ExternalNodes":{"input":"source","output":"drain"},
      "InputParams":{"dc":"ID","ac":0}
    },
    "template":{
      "MasterName":"MosSmallSignalTemplate",
      "ExternalNodes":{
        "gate":"gate","source":"source",
        "drain":"drain","bulk":"bulk"
      },
      "InputParams":{

"GDS":"GDS","CDD":"CDD","CSS":"CSS","CGG":"CGG",
        "CGS":"CGS","CGD":"CGD","GM":"GM","GMB":"GMB"
      }
    }
  }
}
```

Code 3: MosSmallSignalTemplate: Small signal equivalent circuit decomposition

```
"MosSmallSignalTemplate":{
  "ExternalNodes":["gate","source","drain","bulk"],
   "InputParams":["GDS","CDD","CSS","CGG","CGS","CGD",
"GM","GMB"],
  "InternalNodes":[],
  "Schematic":{
    "infr":{
      "MasterName":"resistor",

"ExternalNodes":{"left":"drain","right":"source"},
      "InputParams":{"resistance":1e1000}
    },
    "gds":{
      "MasterName":"ACVCCS",
      "ExternalNodes":{"left":"drain","right":"source",
"input":"drain","output":"source"},
      "InputParams":{"MF":"GDS"}
    },
    "cdd":{
      "MasterName":"capacitor",

"ExternalNodes":{"input":"drain","output":"bulk"},
      "InputParams":{"capacitance":"CDD"}
    },
    "css":{
      "MasterName":"capacitor",

"ExternalNodes":{"input":"source","output":"bulk"},
      "InputParams":{"capacitance":"CSS"}
    },
    "cgg":{
      "MasterName":"capacitor",
      "ExternalNodes":{"input":"gate","output":"bulk"},
      "InputParams":{"capacitance":"CGG"}
    },
    "cgs":{
      "MasterName":"capacitor",

"ExternalNodes":{"input":"gate","output":"source"},
      "InputParams":{"capacitance":"CGS"}
    },
    "cgd":{
      "MasterName":"capacitor",

"ExternalNodes":{"input":"gate","output":"drain"},
      "InputParams":{"capacitance":"CGD"}
    },
    "gm":{
      "MasterName":"ACVCCS",
      "ExternalNodes":{
        "left":"gate","right":"source","input":"drain",
"output":"source"
      },
      "InputParams":{"MF":"GM"}
    },
    "gmb":{
      "MasterName":"ACVCCS",
      "ExternalNodes":{
        "left":"bulk","right":"source","input":"drain",
"output":"source"
      },
      "InputParams":{"MF":"GMB"}
    }
  }
}
```

C Gradient Backpropagation of the Solution of a Linear Equations System

Consider a system of real linear equations $\boldsymbol{A}(\boldsymbol{x})\boldsymbol{v} = \boldsymbol{b}(\boldsymbol{x})$ with respect to $\boldsymbol{v}$, where $\boldsymbol{A}, \boldsymbol{b}$ are nonlinearly dependent on $\boldsymbol{x}$ being a sparse matrix/vector, and $\nabla_{\boldsymbol{x}}\boldsymbol{A}, \nabla_{\boldsymbol{x}}\boldsymbol{b}$ are computable, then $\boldsymbol{v}$ will also be nonlinearly dependent on $\boldsymbol{x}$. Differentiating the system of equations yields:

$$(\boldsymbol{A} + \nabla_{\boldsymbol{x}}\boldsymbol{A} \cdot \mathrm{d}\boldsymbol{x}) \cdot (\boldsymbol{v} + \nabla_{\boldsymbol{x}}\boldsymbol{v} \cdot \mathrm{d}\boldsymbol{x}) = \boldsymbol{b} + \nabla_{\boldsymbol{x}}\boldsymbol{b} \cdot \mathrm{d}\boldsymbol{x}$$

When the zeroth and second order terms are dropped, the following equation holds for any $\mathrm{d}\boldsymbol{x}$

$$\nabla_{\boldsymbol{x}}\boldsymbol{A} \cdot \mathrm{d}\boldsymbol{x} \cdot \boldsymbol{v} + \boldsymbol{A} \cdot \nabla_{\boldsymbol{x}}\boldsymbol{v} \cdot \mathrm{d}\boldsymbol{x} = \nabla_{\boldsymbol{x}}\boldsymbol{b} \cdot \mathrm{d}\boldsymbol{x}, \tag{5}$$

Assume a loss function $l(\boldsymbol{v})$, whose gradient $\nabla_{\boldsymbol{v}}l$ can be calculated. Backpropagating the gradient to $\boldsymbol{x}$ is equivalent to finding the solution of $\nabla_{\boldsymbol{x}}l = \nabla_{\boldsymbol{v}}l \cdot \nabla_{\boldsymbol{x}}\boldsymbol{v}$. In fact, we do not need to actually calculate $\nabla_{\boldsymbol{x}}\boldsymbol{v}$ and store the data, which can be rather dense. According to Equation 5, we have

$$\begin{aligned}&\nabla_{\boldsymbol{x}}\boldsymbol{v} \cdot \mathrm{d}\boldsymbol{x} = \boldsymbol{A}^{-1} \cdot (\nabla_{\boldsymbol{x}}\boldsymbol{b} \cdot \mathrm{d}\boldsymbol{x} - \nabla_{\boldsymbol{x}}\boldsymbol{A} \cdot \mathrm{d}\boldsymbol{x} \cdot \boldsymbol{v}), \forall \mathrm{d}\boldsymbol{x},\\ \Rightarrow &\nabla_{\boldsymbol{v}}l \cdot \nabla_{\boldsymbol{x}}\boldsymbol{v} \cdot \mathrm{d}\boldsymbol{x} = (\nabla_{\boldsymbol{v}}l \cdot \boldsymbol{A}^{-1}) \cdot \big(\nabla_{\boldsymbol{x}}\boldsymbol{b} \cdot \mathrm{d}\boldsymbol{x} - \nabla_{\boldsymbol{x}}\boldsymbol{A} \cdot \mathrm{d}\boldsymbol{x} \cdot \boldsymbol{v}\big), \forall \mathrm{d}\boldsymbol{x},\end{aligned}$$

Therefore, in order to calculate $\nabla_{\boldsymbol{v}}l \cdot \nabla_{\boldsymbol{x}}\boldsymbol{v}$, we only need to solve the sparse matrix linear equations system $\nabla_{\boldsymbol{v}}l \cdot \boldsymbol{A}^{-1}$ once in advance, element-wisely set the values of $\mathrm{d}\boldsymbol{x}$ to 1 and the rest to 0. Then, we obtain $\nabla_{\boldsymbol{x}}l = \nabla_{\boldsymbol{v}}l \cdot \nabla_{\boldsymbol{x}}\boldsymbol{v}$.

For the case of a set of complex linear equations, a similar discussion can also be carried out using the Wirtinger derivative.

MDMMT-2: Multidomain Multimodal Transformer for Video Retrieval, One More Step Towards Generalization

Maksim Dzabraev, Alexander Kunitsyn, Maksim Kalashnikov, Andrei Ivaniuta

Abstract

In this work we present a new state-of-the-art on the text-to-video retrieval task on MSR-VTT, LSMDC, MSVD, YouCook2 and TGIF obtained by a single model. Three different data sources are combined: weakly-supervised videos, crowd-labeled text-image pairs, and text-video pairs. A careful analysis of available pretrained networks helps to choose the best prior-knowledge networks. We introduce three-stage training procedure that provides high transfer knowledge efficiency and allows the usage of noisy datasets during training without prior knowledge degradation. Double positional encoding is used for better fusion of different modalities, and a simple method for non-square inputs processing is suggested. Additionally, we introduce two methods for training multilingual text-to-video models.

Keywords

video, language, retrieval, multimodal, transformer, attention, transfer learning, multilingual

1 Introduction

The text-to-video retrieval task is defined as searching for the most relevant video segments for an arbitrary natural language text query. A search query may contain a description of arbitrary actions, objects, sounds, or a combination of them. Note that an arbitrary search query means zero-shot search. A specific search query might not exist in the training database. Despite this, the model should be able to successfully perform the search operation.

The text-to-video retrieval technology can be used for semantic search within a single long video. For example, inside a full-length movie or a streaming video. After describing the event, the user can easily find the appropriate video segment. A more general task is the search for a relevant video segment within a large video gallery, such as a video hosting platform like YouTube or Vimeo.

Another application is the search for a specific event in a surveillance cameras dataset or a real-time video stream. This can be useful for identifying illegal actions, accidents, or any other important events.

An important requirement for a text-to-video retrieval system is its scalability to a large video gallery. A good example of an efficient architecture is the two-stream models (see Figure 1b). Within this approach the video segment and the text query are encoded independently by the video and text models respectively. Separate processing allows to compute embeddings for the entire video gallery beforehand. During the inference time, the system calculates the embedding for the search query. Next, it calculates the similarity function between query embedding and each embedding from the gallery. The most common choice for similarity function is the cosine similarity. In the case of a single-stream model, visual and text inputs are fused and processed within the same network (see Figure 1a). Because all inputs are joint at the very beginning of the process, such models, in theory, provide stronger interaction between text and video than in multi-stream models. However, there is no known work that demonstrates this. This could be explained by the fact that single-stream models have more capacity and therefore are more likely to overfit to the training dataset. The amount of available public data is not sufficient to prevent this overfitting. A significant downside of the single-stream model for real-world applications is that it does not allow to precompute video embeddings in advance. The data required for the training consists of pairs of video segment and text description. Noise Contrastive Estimation (NCE) is currently the most common framework for this task [14, 33, 37, 28, 13, 6, 15]. Within the framework, the model learns to distinguish a positive pair from a set of negative pairs. The most popular loss functions used in NCE are bi-directional max-margin ranking loss [20] and symmetric cross entropy loss [43, 35, 50]. The mentioned loss functions are based on the following idea. Assume that there is a set of pairs: (text$_k$, video$_k$) where text$_k$ describes video$_k$. Let (u_k, v_k) be the text and video embeddings correspondingly. The objective is to maximize cosine $s_{kk} = \cos(u_k, v_k) = \frac{u_k \cdot v_k}{\|u_k\|_2 \|v_k\|_2}$ for positive pairs and minimize $s_{ij} = \cos(u_i, v_j), i \neq j$ for negative pairs. All s_{ij} form the score matrix, Eq. 1.

$$\begin{array}{c|ccc} & u_1 & \dots & u_n \\ \hline v_1 & s_{11} & \dots & s_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ v_n & s_{n1} & \dots & s_{nn} \end{array} \tag{1}$$

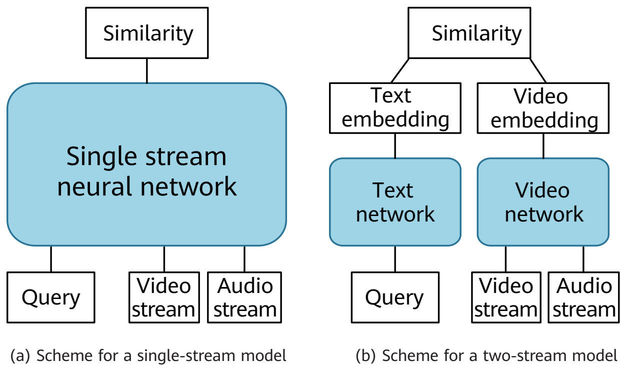
In terms of the score matrix the objective is to maximize diagonal elements and minimize non-diagonal ones.

The bi-directional max-margin ranking loss sets a following optimization problem:

$$\sum_i \sum_{j \neq i} \Big[\max(0, s_{ij} - s_{ii} + m) + \max(0, s_{ji} - s_{ii} + m) \Big] \to \min \tag{2}$$

Parameter m is used to filter out easy examples. The condition $s_{ii} > s_{ij} + m$ means that the positive pair (i, i) and negative pair (i, j) are far enough. Such pairs are counted as easy examples and filtered out. The pairs that are closer to each than m other are used for training. Formally this condition can be represented as

$$\max(0, s_{ij} - s_{ii} + m) \tag{3}$$



(a) Scheme for a single-stream model (b) Scheme for a two-stream model

Figure 1 Two types of fusion

Another popular objective is symmetric cross-entropy loss. Similar to maxmargin ranking loss, it deals with the score matrix and tries to maximize diagonal elements while minimizing non-diagonal ones.

$$-\sum_i \left(\log \frac{\exp(s_{ii}/T)}{\sum_j \exp(s_{ij}/T)} + \log \frac{\exp(s_{ii}/T)}{\sum_j \exp(s_{ji}/T)} \right) \to \min \qquad (4)$$

Because a search query may describe a sound or a visual component, it is important to capture information from both the visual stream and the audio stream of the input video. In this work we fuse information from three modalities: RGB modality (processes each frame independently), motion modality (processes multiple consecutive frames) and audio modality.

2 Related Work

The text-to-video retrieval task is originated in 2016 from the work [40].

Nowadays, there is a large number of high-quality crowd-labeled datasets (captions were created manually by humans) suitable for the text-to-video retrieval task [45, 12, 26, 40, 4, 44, 23, 1, 51, 18], and numerous works using these datasets [15, 6, 13, 28, 19, 11, 14, 46]. In paper [32] the authors leverage large amount of weakly-supervised data (HT100M dataset) from YouTube to train a model. In [14, 11] both weakly-supervised data for pre-training and crowd-labeled datasets for fine-tuning are used.

The task requires a large amount of data, and looking for alternative data sources is reasonable. Because the visual stream of a video is a sequence of frames (images), any individual image can be considered as a one-frame video. In the work [31] the authors successfully use both image-text and video-text datasets.

Impressive results are achieved in the text-to-image retrieval by the Contrastive Language-Image Pre-training (CLIP) model, which is trained with a large amount of web-crawled data [39].

In order to create a text-to-video retrieval model for general application (without specialization for a particular domain) a large amount of data is required. For example, the authors of CLIP used hundreds of millions of data units for their models. Because the video domain is more complex than the image domain, training a general application text-to-video retrieval model presumably requires even more data.

Unfortunately, combining all crowd-labeled text-video and text-image datasets does not allow to approach to the high-quality general application model. In paper [32], the authors attempt to use large amount of weakly-supervised data but the result is still far from that obtained using models trained on crowd-labeled datasets.

It is getting increasingly popular to apply transfer learning-based methods or this task. One of the first successful applications of transfer learning for the text-tovideo retrieval task can be attributed to [30], where several pre-trained networks are used to extract features from video. In the work [14] the authors additionally adopted the Bidirectional Encoder Representations from Transformers (BERT) model [8] as an initialization for the text encoder. Later works [15, 6, 13, 28, 11] use CLIP model as an initialization for both text and vision encoders.

Pre-trained models suitable for the text-to-video ret rieval task can be divided into two classes. The first class is trained using crowd-labeled datasets such as Imagenet [7] or Kinetics [21] datasets. Usually, such models produce taskspecific embeddings, which does not allow to achieve high quality in the text-to-video retrieval task. The second class is trained with a large amount of weaklysupervised data collected from the Internet. The most popular are CLIP, BERT, and irCSN152, which are trained with the IG65M dataset (irCSN152-IG65M) [16].

The analysis of pre-trained models in [11] and our experience show that models trained with a large amount of web-crawled data are able to produce embeddings for general applications and allow to reach better quality in the text-to-video retrieval task.

Using CLIP as an initialization or a feature extractor significantly improves the results in the text-to-video retrieval task [15, 6, 13, 28, 11]. The CLIP model family has several different architectures. All of them have independent text encoders and visual encoders.

In this work we manage to use the text-video, text-image and text-video weakly-supervised (HT100M) datasets together in the same training. In addition, we used the best pre-trained models. This allows us to achieve state-of-the-art results with a single model on many benchmarks.

3 Methodology

Our model follows the idea of MDMMT [14, 11]. However, we suggest an advanced multistage training approach, as well as perform analysis of existing prior knowledge and choose optimal backbones.

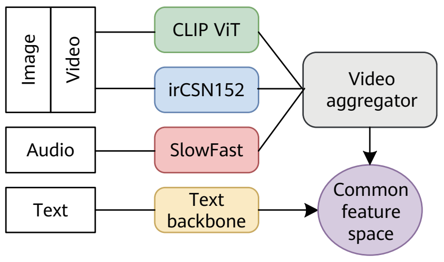


Figure 2 High-level MDMMT-2 architecture. CLIP represents RGB modality, irCSN152 (-IG65M) represents motion modality, SlowFast represents sound modality

3.1 Architecture

The architecture consists of four parts: pre-trained experts, aggregator, text encoder and text embedding projection.

Experts A pre-trained expert is a frozen pre-trained network that produces a sequence of features for input video. In this work we use three experts, each for a different modality. The first one is for image (RGB modality) processes video frames independently. The second one is for motion. It deals with several sequential frames together. The third one is for audio. A general scheme is shown in Figure 2.

For a given video, each expert extracts a sequence $F = [F_1, \dots, F_n]$ of features. For computation efficiency, we limit the video length to 30 seconds, so, for example, if an expert extracts a single feature once every second, a total length n of the feature sequence is not greater than 30.

The features extracted by experts encode the semantics of the video. Each expert outputs features in $\mathbb{R}^{d_{expert}}$. In order to project the different expert features into a common dimension d_{model}, we learn a linear layer for each expert to project all the features into $\mathbb{R}^{d_{model}}$.

Aggregator The aggregator accepts embeddings made by experts and produces a single embedding for each video. A global view of the aggregator is shown in Figure 3 and a detailed view of aggregator's input is shown in Figure 4.

The aggregator follows the architecture of the transformer encoder. It consists of stacked self-attention layers and fully connected layers.

Aggregator's input $\Omega(v)$ is a set of embeddings, all of the same dimension d_{model}. Each of them embeds the semantics of a feature, its modality, and the time in the video when the feature was extracted. This input is given by:

$$\Omega(v) = F(v) + E(v) + T_{start}(v) + T_{end}(v) \tag{5}$$

We define these components below.

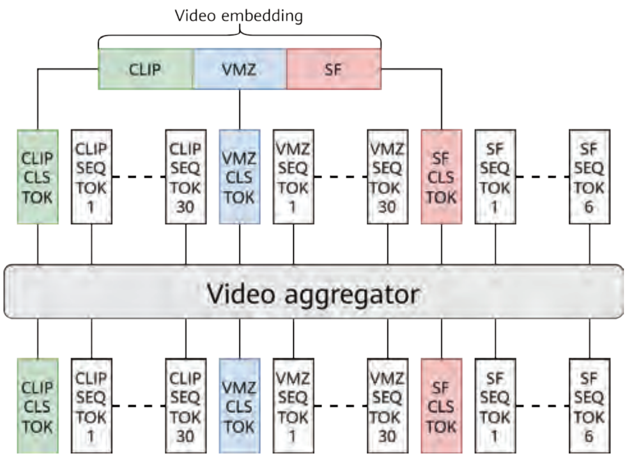


Figure 3 Scheme for video network branch. CLIP represents RGB modality, VMZ (irCSN152-IG65M) represents motion modality, SF (SlowFast) represents sound modality

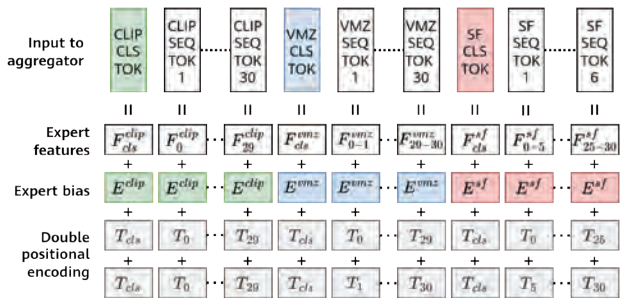


Figure 4 A detailed scheme of aggregator's input. CLIP represents RGB modality, VMZ (irCSN152-IG65M) represents motion modality, SF (SlowFast) represents sound modality

Features F The aggregator produces an embedding for each of its feature inputs, resulting in several embeddings for an expert. In order to obtain a unique embedding for each expert, we define an aggregated embedding F_{cls}^{expert} (expert CLS token) that will collect and contextualize the expert's information. We initialize this embedding with a max pooling aggregation of all the corresponding expert's features. The sequence of input features to the aggregator then takes the form:

$$F(v) = [F_{cls}^{clip}, F_0^{clip}, \dots, F_{29}^{clip}, F_{cls}^{vmz}, F_{0-1}^{vmz}, \dots, F_{29-30}^{vmz}, F_{cls}^{sf}, F_{0-5}^{sf}, \dots, F_{25-30}^{sf}] \tag{6}$$

Expert embeddings E In order to process cross-modality information, our aggregator needs to identify which expert it is attending to. We learn embeddings E^{expert} of size d_{model} to distinguish between embeddings of different experts. Thus, the sequence of expert embeddings to our video encoder takes the form:

$$E(v) = [E^{clip}, E^{clip}, \dots, E^{vmz}, E^{vmz}, \dots, E^{sf}, E^{sf}], \dots \tag{7}$$

Double positional encoding $T_{start} + T_{end}$ Each expert takes a different type and shape of data as input. For example, CLIP takes a single image frame to produce an embedding. irCSN152-IG65M (VMZ) produces a single embedding from a sequence of 32 consecutive frames (32 fps). Slow-Fast (SF) [22] takes a Mel spectrogram of a 5 seconds long audio frame to produce an embedding.

Positional encoding is used in the transformer encoder architecture to provide information about the order of tokens in the input sequence. In our case, the positional (temporal) encoding has to provide information not only about the order of tokens but also about the time length of each individual token.

We introduce double positional encoding: for each embedding we add two biases in $\mathbb{R}^{d_{model}}$. The first bias $T_{start}(v)$ stands for the beginning timestamp of the video segment and the second one $T_{end}(v)$ stands for the ending timestamp.

This way we ensure that different time lengths per expert embedding are processed correctly. The results in Table 2 (Section 4) support this novelty when compared with the single temporal encoding used in previous works [14, 11], where by single temporal encoding we mean averaging timestamps of the beginning and the ending of a segment, instead of using them separately.

We also learn additional temporal embeddings T_{cls} which encode aggregated features. The sequence of temporal embeddings then takes the form:

$$T_{start}(v) = [T_{cls}, T_0, \dots, T_{29}, T_{cls}, T_0, \dots, T_{29}, T_{cls}, T_0, \dots, T_{25}] \quad (8)$$

$$T_{end}(v) = [T_{cls}, T_0, \dots, T_{29}, T_{cls}, T_1, \dots, T_{30}, T_{cls}, T_5, \dots, T_{30}] \quad (9)$$

Video embedding Final video embedding is a concatenation of expert CLS tokens, processed by the aggregator (see Figure 3). Given N experts, video embedding size is $N \times d_{model}$.

Text encoder and text embedding projection The text encoder takes arbitrary English natural language text and produces embedding in $\mathbb{R}^{d_{text}}$. In order to match the size d_{text} of the text embedding with video embedding, the text embedding projection part maps the text embedding to the distinct space $\mathbb{R}^{d_{model}}$ for each modality (see Figure 5). For projection we use the gated embedding module [31], which consists of fully-connected, activation and normalization layers.

Each projection of size d_{model} is then scaled by a mixture weight w_{expert} (one scalar weight per expert projection), which is computed by applying a single linear layer to the text embedding, and passing the result through a softmax to ensure that the mixture weights total to 1. For example, given a query "a man is talking", the biggest weight is assigned to audio modality, whereas for a query "a man shakes his head", the model focuses most on motion modality.

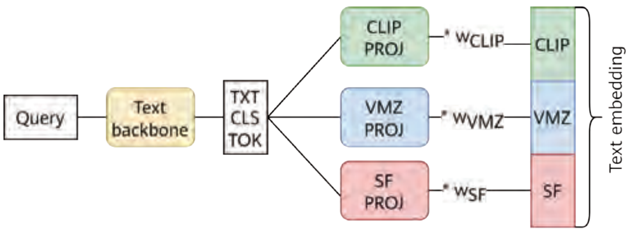


Figure 5 Scheme for text network branch. CLIP represents RGB modality, VMZ (irCSN152-IG65M) represents motion modality, SF (SlowFast) represents sound modality

Final text embedding is a concatenation of expert projections, multiplied by corresponding mixture weights. Given N experts, video embedding size is $N \times d_{model}$.

Note that this architecture is flexible. It is possible to add or remove additional modalities. Also, it is possible to replace a given pre-trained text encoder with another one. For example, it is possible to use CLIP ViT-B/32 as an RGB expert and the text part of CLIP ViT-B/16 as a text encoder.

Table 1 "Num videos" is the number of video clips (images) in the dataset, "Num pairs" is the total number of video-caption (image-caption) pairs, and "Num unique captions" is the number of unique captions in the dataset

Dataset	Num videos (images)	Num pairs	Num unique captions
MSR-VTT [45]	10k	200k	167k
ActivityNet [12]	14k	70k	69k
LSMDC [40]	101k	101k	101k
TwitterVines [1]	6.5k	23k	23k
YouCook2 [51]	1.5k	12k	12k
MSVD [4]	2k	80k	64k
TGIF [26]	102k	125k	125k
SomethingV2 [18]	193k	193k	124k
VATEX [44]	28k	278k	278k
TVQA [23]	20k	179k	178k
Sum above	**477k**	**1261k**	
Flicker30k [47]	32k	159k	158k
COCO [5]	123k	617k	592k
Conceptual Captions [42]	3M	3M	2M

3.2 Datasets

A list of datasets used in this work is provided in Table 1. Only training splits of the listed datasets are used in the training dataset. Note that we use both text-video and text-image datasets. In Section 4.4 we show results for video-only datasets and image+video datasets. Since each dataset has a different number of videos and captions, it is important to combine datasets properly [11].

In the following experiments MSR-VTT full clean split is used. This split is introduced in [11]. The test part of a full clean split is the same as the test part of a full split. The training part of a full clean split is mostly similar to a full split but some videos are removed. All removed videos have a corresponding duplicate in the test part.

3.3 Loss

The MDMMT-2 is trained with the bi-directional max-margin ranking loss [20] (see Equation 2).

In our experiments we compare bi-directional max-margin ranking loss and symmetric cross-entropy loss. Both objectives show the same result, but the first objective converges faster.

Table 2 Comparison of a standard positional encoding with the proposed double positional encoding. Dataset: MSR-VTT full clean split; Text backbone: CLIP ViT-B/32; Experts: CLIP ViT-L/14, irCSN152-IG65M, SF

Temporal Embedding	Text → Video			
	R@1↑	R@5↑	R@10↑	MdR↓
Single	$22.1_{\pm0.1}$	$48.2_{\pm0.0}$	$60.0_{\pm0.1}$	$6.0_{\pm0.0}$
Double	$22.2_{\pm0.1}$	$48.5_{\pm0.2}$	$60.3_{\pm0.2}$	$6.0_{\pm0.0}$

4 Experiments

In sections 4.1 - 4.3 all experiments are made on the MSR-VTT full clean split (see Section 3.2) for 50 epochs and 60k examples per epoch. The initial learning rate is 5e-5. After each epoch we multiply the learning rate by $\gamma = 0.95$. In these experiments we freeze the text backbone and train only the aggregator model and the text embedding projection part.

For the MSR-VTT training we use an aggregator with 4 layers and 4 heads. On a larger dataset (see Section 4.4 - 4.6) the aggregator has 9 layers and 8 heads. We set $d_{model} = 512$ in all experiments.

Results are reported as mean$_{\pm std}$ or just mean over 3 experiments. By R@k, MnR, MdR we denote recall at k, mean rank, and median rank correspondingly.

4.1 CLIP

In paper [11], it is shown that CLIP works as a strong visual feature extractor and outperforms other available models by a large margin. We found out that the CLIP text backbone also works better than other available text models, such as BERT [8], which was originally used in [14], or GPT [3].

Currently there are several publicly available CLIP models. In this section we compare their performance to make sure that we use the best possible combination. Results are presented in Table 3.

Our observations:

- Suppose we have a pre-trained CLIP: text backbone and the corresponding visual backbone. We observe that if we replace the original visual backbone with a bigger/deeper one, we obtain a better video retrieval system.

Table 3 Comparison of CLIP visual and text backbones combinations. Experts: CLIP; Metric: R@5

Visual \ Text	RN50	RN50x4	RN50x16	RN50x64	ViT-B/32	ViT-B/16	ViT-L/14
RN50	40.1	38.7	39.3	39.3	40.1	39.8	39.8
RN50x4	42.8	41.9	42.5	42.5	43.2	43.1	43.2
RN50x16	43.9	43.5	43.6	43.0	44.4	44.5	44.4
RN50x64	**44.6**	43.9	44.1	44.2	44.8	45.2	45.4
ViT-B/32	42.0	41.2	40.9	40.9	42.5	42.4	42.2
ViT-B/16	44.4	43.8	43.4	43.3	44.8	45.4	44.9
ViT-L/14	46.2	45.7	45.3	45.3	46.5	46.8	**47.2**

Table 4 Experts combinations. Text backbone: CLIP ViT-B/32

Experts			Text → Video		
CLIP	irCSN152-IG65M	SF	R@1↑	R@5↑	MdR↓
	√		$10.2_{\pm0.0}$	$29.3_{\pm0.1}$	$17.3_{\pm0.5}$
	√	√	$11.2_{\pm0.1}$	$31.5_{\pm0.2}$	$15.0_{\pm0.0}$
√			$21.3_{\pm0.1}$	$46.5_{\pm0.2}$	$7.0_{\pm0.0}$
√	√		$21.5_{\pm0.1}$	$46.7_{\pm0.1}$	$7.0_{\pm0.0}$
√		√	$22.0_{\pm0.1}$	$47.8_{\pm0.1}$	$6.0_{\pm0.0}$
√	√	√	**$22.2_{\pm0.1}$**	**$48.5_{\pm0.2}$**	**$6.0_{\pm0.0}$**

- If we use the same visual backbone with different text backbones, a text backbone of a bigger/deeper model does not necessarily show better results. Tab. 3 demonstrates that among residual neural networks (ResNets) the best result (in bold) is achieved with a combination of the deepest visual backbone (RN50x64) and the text backbone from the most shallow model (RN50).
- CLIP ViT-L/14 shows the overall best performance both as a visual backbone and a text backbone (in bold).

4.2 Experts Combination

Using combination of different experts allows to achieve a better performance. In Table 4 various combinations of experts are presented. Usage of all three modalities gives the best result.

4.3 Dealing with Non-square Videos

Both irCSN152-IG65M and CLIP take videos (images) of square shape as input. Therefore, it is not possible to directly use information from the whole video frame. It may happen that objects or actions are taking place in the corner (out of the center crop) of the video. If we use center crop to compute embeddings, the information from the corners will be lost. There are several possible solutions to this problem:

- Squeeze a video to a square without saving the aspect ratio (*squeeze*)
- Pad a video to a square with blackbars (*padding*)
- Take several crops from the video, average the embeddings of these crops, and use this average as the embedding (*mean*)

For the *mean* technique we take three crops: left or bottom, center, right or top (depending on the video orientation) and then average the embeddings of these crops.

Table 5 Comparison of different techniques for extracting features from nonsquare videos. Text backbone: CLIP ViT-B/32; Experts: CLIP ViT-L/14; Metric: R@5

Text / Train	Squeeze	Center crop	Padding	Mean
Squeeze	46.3	46.0	46.0	**47.1**
Center Crop	46.0	46.5	46.0	**47.3**
Padding	46.0	46.2	46.7	**47.0**
Mean	45.9	46.4	45.9	**47.4**

Experiments in Table 5 show that *squeeze* works worse than the center crop, *padding* works slightly better than the center crop, and *mean* works the best.

We want to emphasize that using *mean* during test improves video-retrieval performance even if other methods are used during training.

4.4 Adding Images

In paper [11], it is shown that the proper combination of datasets allows to train a single model that can capture the knowledge from all used datasets. In most cases the model trained on the combination of datasets is better than the model trained on a single dataset.

In Table 7 we show that a proper combination of text-video and text-image datasets allows to improve video-retrieval performance. Hyperparameters are specified in Section 4.5, stage S_1.

Weights that are used to combine all datasets are specified in Table 6. First 10 rows are video datasets (denoted as 10V) and last 3 are image datasets (denoted as 3I). A weight for each dataset is proportional to the number and quality of video-caption pairs in this dataset, though there is no strict formula and small adjustments to the weights values do not significantly change the result.

4.5 Pre-training and Fine-tuning

Note that in our work the aggregator is initialized from scratch, whereas text backbone is pre-trained. If we simultaneously train a randomly initialized aggregator and a pre-trained text backbone, then during the time the aggregator is trained, the text backbone might degrade. This is why, for the final result we introduce a training procedure that consists of three stages (denoted as S_0, S_1, S_2).

During stage S_0, we use the noisy HT100M dataset. The text backbone is frozen, and only the aggregator and text embedding projection part are trained.

Table 6 Datasets used in train procedure. "Weight" describes how often we sample examples from the dataset. The probability of obtaining an example from the dataset with the weight w equals to w divided by the sum of all weights

Dataset	Weight	Type
MSR-VTT	140	Text-video datasets (10V)
ActivityNet	100	
LSMDC	70	
Twitter Vines	60	
YouCook2	20	
MSVD	20	
TGIF	102	
SomethingV2	169	
VATEX	260	
TVQA	150	
COCO	280	Text-image datasets (3I)
Flicker30k	200	
Conceptual Captions	160	

Table 7 Test results on MSR-VTT full clean split. Text backbone: CLIP ViTB/32; Experts: CLIP ViT-L/14, irCSN152-IG65M, SF

Dataset	Text → Video			
	R@1↑	R@5↑	R@10↑	MdR↓
10V	30.2	56.6	67.1	4.0
10V+3I	30.9	57.4	67.8	4.0

During stage S_1, we use crowd-labeled datasets 10V+3I. Same as in S_0, the text backbone is frozen, and only the aggregator and the text embedding projection part are trained.

During stage S_2, same as in S_1, we use crowd-labeled datasets 10V+3I. Now, however, we unfreeze the text backbone and train all three main components: aggregator, text backbone and text embedding projection.

Hyperparameters for these stages are listed in Table 8. They were selected in accordance with a following idea – we want to extract as much knowledge as possible till the model does not start to overfit. Results for different combinations of stages are listed in Table 9.

4.6 Final Result

In this section we compare our solution with the prior art. Our best solution uses three modalities: CLIP ViT-L/14 (RGB modality), irCSN152-IG65M (motion modality), Slow-Fast trained on VGG-Sound (audio modality). Text backbone is used from CLIP ViT-L/14. To fuse modalities we use an aggregator with 9 layers and 8 heads. The training procedure is described in Section 4.5. Results are shown in Table 10 - Table 15.

Table 8 Hyperparameters for different stages

Train stage	Examples per epoch	Num. epochs	Learning rate	γ	Datasets
S_0	60k	200	5e-5	0.98	HT100M
S_1	380k	45	5e-5	0.95	10V+3I
S_2	200k	20	2e-5	0.8	10V+3I

Table 9 Test results for train stages on MSR-VTT full clean split. Text backbone: CLIP ViT-B/32; Experts: CLIP ViT-L/14, irCSN152-IG65M, SF

Train stages			Text → Video		
S_0	S_1	S_2	R@1↑	R@5↑	MdR↓
√			7.7	19.0	60.0
	√		29.0	55.3	4.0
	√	√	30.5	56.9	4.0
√	√		31.2	57.8	4.0
√	√	√	32.5	59.4	3.0

The center crop is used for visual features extraction during training and testing for all datasets except MSR-VTT (see Table 11), where we report two results on testing set: center crop and *mean* methods (see Section 4.3). On other datasets, *mean* technique didn't bring significant improvement. This is most probably due to the fact that MSR-VTT has 60k captions in its test set, which is by an order of magnitude more than in other test sets used in this work. So, in other datasets, there are just not enough captions with the required context to see the improvement.

In order to avoid repetitive decoding of the same videos, we first extract all required expert embeddings and train our model on these embeddings instead of raw videos.

Results on MSR-VTT, LSMDC, MSVD, YouCook2, TGIF are obtained by using a single model. Our model outperforms SOTA by 1.6%, 0.6%, 3.9%, 4.3%, 1.1% correspondingly on R@5. On MSR-VTT-1kA (see Table 10) we report two results with different training splits: full(7k) and 1k-A(9k). The first result approaches SOTA and the second result outperforms SOTA by 0.8% on R@5.

Table 10 Test results on the MSR-VTT-1k-A dataset. Results that were obtained using original testing protocol (without dual softmax [6, 15] on inference) are shown. Results are collected from articles and https://paperswithcode.com/sota/video-retrieval-on-msr-vtt-1ka

Model	**MSR-VTT-1k-A text → video**				
	R@1↑	**R@5↑**	**R@10↑**	**MnR↓**	**MdR↓**
JSFusion [48]	10.2	31.2	43.2	—	13.0
E2E [32]	9.9	24.0	32.4	—	29.5
HT [33]	14.9	40.2	52.8	—	9.0
CE [27]	20.9	48.8	62.4	28.2	6.0
CLIP [38]	22.5	44.3	53.7	61.7	8.0
MMT [14]	26.6	57.1	69.6	24.0	4.0
AVLnet[41]	27.1	55.6	66.6	—	4.0
SSB [36]	30.1	58.5	69.3	—	3.0
CLIP agg [37]	31.2	53.7	64.2	—	4.0
MDMMT [11]	38.9	69.0	79.7	16.5	2.0
CLIP4Clip [28]	44.5	71.4	81.6	15.3	2.0
CLIP2Video [13]	45.6	72.6	81.7	14.6	2.0
LAFF [19]	45.8	71.5	82.0	—	—
CAMoE [6]	44.6	72.6	81.8	**13.3**	2.0
MDMMT-2 full (Ours)	$46.5_{\pm0.8}$	$74.3_{\pm0.6}$	$\mathbf{83.3}_{\pm0.2}$	$14.1_{\pm0.1}$	$2.0_{\pm0.0}$
QB-Norm+CLIP2Video [2]	47.2	73.0	83.0	—	2.0
CLIP2TV [15]	48.3	74.6	82.8	14.9	2.0
MDMMT-2 1k-A (Ours)	$\mathbf{48.5}_{\pm0.3}$	$\mathbf{75.4}_{\pm0.3}$	$\mathbf{83.9}_{\pm0.5}$	$13.8_{\pm0.3}$	$\mathbf{2.0}_{\pm0.0}$

Table 11 Test results on the MSR-VTT dataset. Results are collected from articles and https://paperswithcode.com/sota/video-retrieval-on-msr-vtt

Model	**Split**	**MSR-VTT text → video**				
		R@1↑	**R@5↑**	**R@10↑**	**MnR↓**	**MdR↓**
VSE [34]	full	5.0	16.4	24.6	—	47.0
VSE++ [34]		5.7	17.1	24.8	—	65.0
Multi Cues [34]		7.0	20.9	29.7	—	38.0
W2VV [9]		6.1	18.7	27.5	—	45.0
Dual Enc. [10]		7.7	22.0	31.8	—	32.0
CE [27]		10.0	29.0	41.2	86.8	16.0
MMT [14]		10.7	31.1	43.4	88.2	15.0
CLIP [38]		15.1	31.8	40.4	184.2	21.0
CLIP agg [37]		21.5	41.1	50.4	—	4.0
MDMMT [11]		23.1	49.8	61.8	52.8	6.0
TACo [46]		24.8	52.1	64.0	—	5.0
LAFF [19]		29.1	54.9	65.8	—	—
CLIP2Video [13]		29.8	55.5	66.2	45.4	4.0
CAMoE [6]		32.9	58.3	68.4	42.6	3.0
CLIP2TV [15]		33.1	58.9	68.9	44.7	3.0
MDMMT-2 (Ours)		$\mathbf{33.4}_{\pm0.1}$	$\mathbf{60.1}_{\pm0.1}$	$\mathbf{70.5}_{\pm0.1}$	$\mathbf{39.2}_{\pm0.2}$	$\mathbf{3.0}_{\pm0.0}$
MDMMT-2 test mean (Ours)		$\mathbf{33.7}_{\pm0.1}$	$\mathbf{60.5}_{\pm0.0}$	$\mathbf{70.8}_{\pm0.1}$	$\mathbf{37.8}_{\pm0.3}$	$\mathbf{3.0}_{\pm0.0}$
MMT [14]	full clean	10.4	30.2	42.3	89.4	16.0
MDMMT [11]		10.4	49.5	61.5	53.8	6.0
MDMMT-2 (Ours)		**33.3**	**59.8**	**70.2**	**38.7**	**3.0**

Table 12 Test results on the LSMDC dataset. Results are collected from articles and https://paperswithcode.com/sota/video-retrieval-on-lsmdc

Model	LSMDC text → video				
	R@1↑	R@5↑	R@10↑	MnR↓	MdR↓
CT-SAN [49]	5.1	16.3	25.2	—	46.0
JSFusion [48]	9.1	21.2	34.1	—	36.0
MEE [31]	9.3	25.1	33.4	—	27.0
MEE-COCO [31]	10.1	25.6	34.6	—	27.0
CE [27]	11.2	26.9	34.8	96.8	25.3
CLIP agg [37]	11.3	22.7	29.2	—	56.5
CLIP [38]	12.4	23.7	31.0	142.5	45.0
MMT [14]	12.9	29.9	40.1	75.0	19.3
MDMMT [11]	18.8	38.5	47.9	58.0	12.3
CLIP4Clip [28]	21.6	41.8	49.8	58.0	—
QB-Norm+CLIP4Clip [2]	22.4	40.1	49.5	—	11.0
CAMoE [6]	25.9	46.1	53.7	54.4	—
MDMMT-2 (Ours)	**26.9**$_{\pm 0.6}$	**46.7**$_{\pm 0.5}$	**55.9**$_{\pm 0.4}$	**48.0**$_{\pm 0.5}$	**6.7**$_{\pm 0.5}$

Table 13 Test results on the MSVD dataset. Results are collected from articles and https://paperswithcode.com/sota/video-retrieval-on-msvd

Model	MSVD text → video				
	R@1↑	R@5↑	R@10↑	MnR↓	MdR↓
LAFF [19]	45.4	76.0	84.6	—	—
CLIP4Clip [28]	46.2	76.1	84.6	10.0	2.0
CLIP2Video [13]	47.0	76.8	85.9	9.6	2.0
QB-Norm+CLIP2Video [2]	48.0	77.9	86.2	—	2.0
CAMoE [6]	49.8	79.2	87.0	9.4	—
MDMMT-2 (Ours)	**56.8**$_{\pm 0.2}$	**83.1**$_{\pm 0.2}$	**89.2**$_{\pm 0.1}$	**8.8**$_{\pm 0.0}$	**1.0**$_{\pm 0.0}$

Table 14 Test results on the YouCook2 dataset. Results are collected from articles and https://paperswithcode.com/sota/video-retrieval-on-youcook2

Model	YouCook2 text → video				
	R@1↑	R@5↑	R@10↑	MnR↓	MdR↓
Text-Video Embedding [33]	8.2	24.5	35.3	—	24.0
COOT [17]	16.7	—	52.3	—	—
UniVL [29]	28.9	57.6	70.0	—	4.0
TACo [46]	29.6	59.7	72.7	—	4.0
MDMMT-2 (Ours)	**32.0**$_{\pm 0.7}$	**64.0**$_{\pm 0.3}$	**74.8**$_{\pm 0.2}$	**12.7**$_{\pm 0.3}$	**3.0**$_{\pm 0.0}$

Table 15 Test results on the TGIF dataset. Results are collected from articles and https://paperswithcode.com/sota/video-retrieval-on-tgif

Model	TGIF text → video				
	R@1↑	R@5↑	R@10↑	MnR↓	MdR↓
W2VV++ [25]	9.4	22.3	29.8	—	—
SEA [24]	11.1	25.2	32.8	—	—
LAFF [19]	24.5	45.0	54.5	—	—
MDMMT-2 (Ours)	**25.5**$_{\pm 0.1}$	**46.1**$_{\pm 0.0}$	**55.7**$_{\pm 0.1}$	**94.1**$_{\pm 0.3}$	**7.0**$_{\pm 0.0}$

5 Multilingual Text Backbone

In this section we present two approaches for creating multilingual models:

- Multilingual text-video training database
- Text backbone distillation (applicable only for two-stream models)

5.1 Multilingual Training Database

Most of the text-to-video datasets use English-only captions. Modern translation systems allow to translate the caption from English to almost any other language with decent quality, so it is possible to translate each caption in the dataset and thus create a multilingual training database.

If we use several languages, it is required to use a multilingual tokenizer, which, in turn, requires us to change the pretrained text backbone. For example, if bert-base-uncased model is used for English-only captions, then in order to work with multilingual captions we need to switch to bert-base-multilingual-uncased with the corresponding multilingual tokenizer. Although, if we use the CLIP text backbone, this method cannot be applied as there are no publicly available multilingual CLIP models.

Consequently, the same training procedure can be applied for both monolingual and multilingual datasets. It requires translating dataset captions to desired languages and replacing both the tokenizer and the text backbone.

Figure 6 shows that training on two languages shows a slight decrease in the original language (EN) and a two-percent decrease in the translated language (ZH). This decrease occurs because of machine translation, which is not perfectly accurate and may introduce some noise.

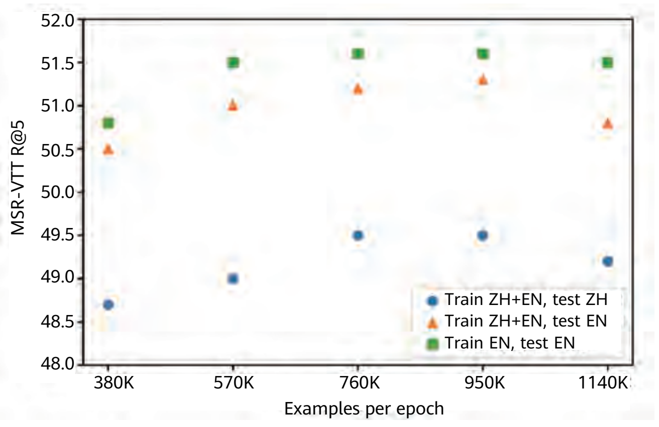


Figure 6 Experiments on multilingual training database. Text backbone: bertbase- multilingual-uncased; Experts: CLIP ViT-B/32, irCSN152-IG65M, SlowFast. Both video aggregator and text backbone are trained in a setting similar to S1 (see Table 8)

5.2 English Model Knowledge Distillation

The two-stream architecture allows to apply the distillation procedure to the text model. Within this approach, the visual part does not change. Instead, a new text model is trained for the existing visual part. This model accepts texts in target languages as input. The original model is called a teacher and the new model for the target languages is called a student. The method does not require retraining of the visual backbone, which means that no recalculation of precomputed visual embeddings is required. This leads to the original model distillation possibility with few computational resources.

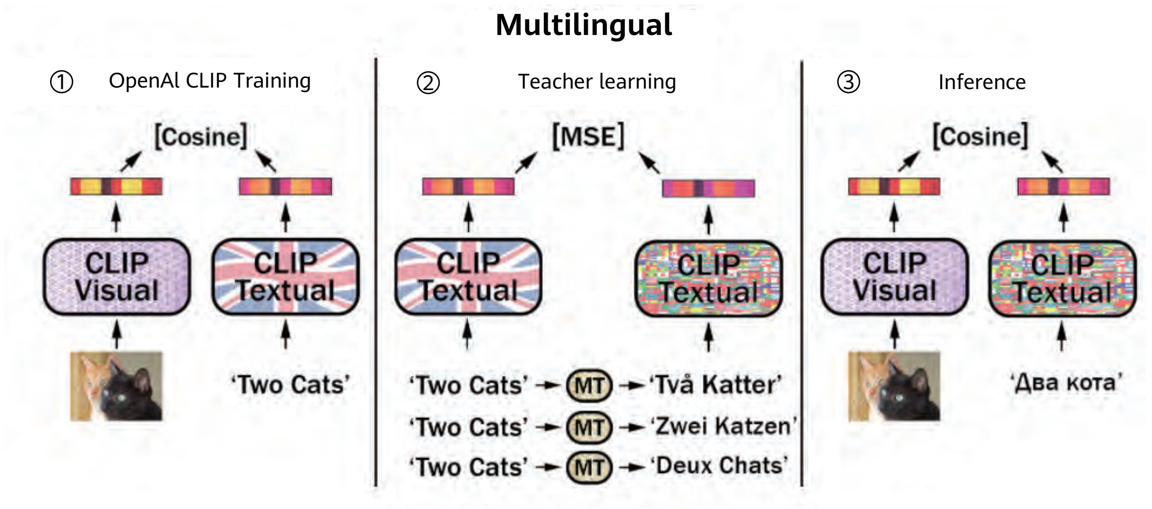


Figure 7 Multilingual text backbone distillation. Source: https://github.com/FreddeFrallan/Multilingual-CLIP

There are no strict requirements for the student model architecture, initialization, or tokenizer as long as it is applicable for target languages. The only restriction is that the text query must be encoded into a single embedding with a shape similar to the teacher model embedding.

The training procedure is performed with the teacher model frozen. The text text_{src} from the original dataset is selected randomly. Then the corresponding translation to the target language is text_{tgt}. text_{src} is processed by the teacher model and text_{tgt} by the student model. Next, we apply Mean Squared Error (MSE) loss to the obtained embeddings (teacher embedding is the ground truth).

$$\sum_k (\text{teacher}(\text{text}_{\text{src}})_k - \text{student}(\text{text}_{\text{tgt}})_k)^2 \to \min \qquad (10)$$

The dataset, which was used for the original system training, can be considered as a good dataset for the distillation. Additionally, we want to highlight that the student model cannot outperform the teacher because the teacher model output is the only training signal.

In this work the teacher model understands only English input. The target languages are English and Chinese. The Chinese texts are obtained by a machine translation system. The quality is measured with the original MSR-VTT dataset (English) and translated into Chinese.

In our experiments the teacher model is a transformer encoder network with 12 layers, 8 heads, and width equal to 512 (based on CLIP ViT-B/16). The student model has the same architecture with a different tokenizer (for English and Chinese) and a different number of token embeddings.

As we can see in Table 16, which represents our distillation results, the performance of the student model on the original English MSR-VTT benchmark almost reached the performance of the teacher model. The result of the student model on MSR-VTT translated to Chinese is worse by 2%. This performance decrease can be explained by the fact that the machine translation system does not work perfectly and produces errors.

Table 16 Knowledge distillation

Model	**MSR-VTT → video**	
	R@5↑ EN	**R@5↑ ZH**
teacher	57.3	—
student ENZH	57.12	55.3

6 Conclusions

We performed a refined study of each conceptual part of the transformer application for the text-to-video retrieval task. The analysis of the prior knowledge allows to choose optimal existing backbone experts. The combination of different types of data sources allows to significantly increase the overall training data amount. Also we suggest a multi-stage training procedure without experts fine-tuning, which prevents their overfitting on a particular domain. The usage of expanded data and optimal experts leads to a great increase in the generalization ability. It allows to obtain a model, which simultaneously performs well in multiple domains and benefits with the growth of domains diversity. We demonstrate a significant novelty – a possibility to obtain SOTA results in different domains by using the same model, instead of preparing a domain-specific model for each domain. In particular, we obtained new SOTA results in MSR-VTT, LSMDC, MSVD, YouCook2 and TGIF with a single model that was trained only once.

Two different approaches for training multilingual models are suggested. We show that it is possible to obtain a multilingual retrieval model with almost indistinguishable performance on the original language and competitive performance on target languages.

References

[1] George Awad *et al.*, "TRECVID 2020: comprehensive campaign for evaluating video retrieval tasks across multiple application domains," in *Proceedings of TRECVID 2020*, NIST, USA, 2020.

[2] Simion-Vlad Bogolin *et al.*, "Cross modal retrieval with querybank normalisation," 2021, arXiv: 2112.12777 [cs.CV].

[3] Tom B. Brown *et al.*, "Language models are few-shot learners," 2020, arXiv: 2005.14165 [cs.CL].

[4] David Chen and William Dolan, "Collecting highly parallel data for paraphrase Evaluation," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA: Association for Computational Linguistics, 2011, pp. 190–200.

[5] Xinlei Chen *et al.*, "Microsoft COCO captions: data collection and evaluation server," 2015, arXiv: 1504.00325 [cs.CV].

[6] Xing Cheng *et al.*, "Improving video-text retrieval by multi-stream corpus alignment and dual softmax loss," 2021, arXiv: 2109.04290 [cs.CV].

[7] J. Deng *et al.*, "ImageNet: a large-scale hierarchical image database," in CVPR09, 2009.

[8] Jacob Devlin *et al.*, "BERT: pre-training of deep bidirectional transformers for language understanding," 2018, arXiv: 1810.04805.

[9] Jianfeng Dong, Xirong Li, and Cees G. M. Snoek, "Predicting visual features from text for image and video caption retrieval," in *IEEE Transactions on Multimedia 20.12 (2018)*, pp. 3377–3388. *issn: 1941-0077, doi:10.1109/tmm.2018.2832602*.

[10] Jianfeng Dong *et al.*, "Dual encoding for zero-example video retrieval," 2019, arXiv: 1809.06181 [cs.CV].

[11] Maksim Dzabraev *et al.*, "MDMMT: multidomain multimodal transformer for video retrieval," 2021, doi: 10.1109/cvprw53098.2021.00374.

[12] Bernard Ghanem Fabian Caba Heilbron Victor Escorcia and Juan Carlos Niebles, "ActivityNet: a large-scale video benchmark for human activity understanding," 2015.

[13] Han Fang *et al.*, "CLIP2Video: mastering video-text retrieval via image CLIP," 2021.

[14] Valentin Gabeur *et al.*, "Multi-modal transformer for video retrieval," 2020, arXiv: 2007.10639 [cs.CV].

[15] Zijian Gao *et al.*, "CLIP2TV: an empirical study on transformer-based methods for video-Text retrieval," 2021, arXiv: 2111.05610 [cs.CV].

[16] Deepti Ghadiyaram *et al.*, "Large-scale weakly-supervised pre-training for video action recognition," 2019, arXiv: 1905.00561 [cs.CV].

[17] Simon Ging *et al.*, "COOT: cooperative hierarchical transformer for video-text representation learning," in *CoRR abs/2011.00597 (2020)*, arXiv: 2011.00597.

[18] Raghav Goyal *et al.*, "The "something something" video database for learning and evaluating visual common sense," 2017, arXiv: 1706.04261 [cs.CV]. MDMMT-2 23

[19] Fan Hu *et al.*, "Lightweight attentional feature fusion for video retrieval by text," 2021, arXiv: 2112.01832 [cs.MM].

[20] Andrej Karpathy, Armand Joulin, and Li Fei-Fei, "Deep fragment embeddings for bidirectional image sentence mapping," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, Cambridge, MA, USA: MIT Press, 2014, 1889–1897.

[21] Will Kay *et al.*, "The kinetics human action video dataset," 2017, arXiv: 1705.06950 [cs.CV].

[22] Evangelos Kazakos *et al.*, "Slow-fast auditory streams for audio recognition," 2021, arXiv: 2103.03516 [cs.SD].

[23] Jie Lei *et al.*, "TVQA: localized, compositional video question answering," 2019, arXiv: 1809.01696 [cs.CL].

[24] Xirong Li *et al.*, "SEA: sentence encoder assembly for video retrieval by textual queries," in *IEEE Transactions on Multimedia 23 (2021)*, pp. 4351–4362. doi: 10.1109/TMM.2020.3042067.

[25] Xirong Li *et al.*, "W2VV++: fully deep learning for ad-hoc video search." 2019, doi: 10.1145/3343031.3350906.

[26] Yuncheng Li *et al.*, "TGIF: a new dataset and benchmark on animated GIF description," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[27] Yang Liu *et al.*, "Use what you have: video retrieval using representations from collaborative experts," 2020, arXiv: 1907.13487 [cs.CV].

[28] Huaishao Luo *et al.*, "CLIP4Clip: an empirical study of CLIP for end to end video clip retrieval," 2021.

[29] Huaishao Luo *et al.*, "UniViLM: a unified video and language pre-training model for multimodal understanding and generation," in *CoRR abs/2002.06353 (2020)*, arXiv: 2002.06353.

[30] Antoine Miech, Ivan Laptev, and Josef Sivic, "Learning a text-video embedding from incomplete and heterogeneous data," in *arXiv preprint*, arXiv:1804.02516 (2018).

[31] Antoine Miech, Ivan Laptev, and Josef Sivic, "Learning a text-video embedding from incomplete and heterogeneous data," 2020, arXiv: 1804.02516 [cs.CV].

[32] Antoine Miech *et al.*, "End-to-end learning of visual representations from uncurated instructional videos," 2020, arXiv: 1912.06430 [cs.CV].

[33] Antoine Miech *et al.*, "HowTo100M: learning a text-video embedding by watching hundred million narrated video clips," in *ICCV*, 2019.

[34] Niluthpol Chowdhury Mithun *et al.*, "Learning joint embedding with multimodal cues for cross-modal video-text retrieval," in *Proceedings of the 2018 ACM on International Conference on Multimedia Retrieval*, 2018, pp. 19–27.

[35] Aäon van den Oord, Yazhe Li, and Oriol Vinyals, "Representation learning with contrastive predictive coding," in *CoRR abs/1807.03748 (2018)*, arXiv: 1807.03748.

[36] Mandela Patrick *et al.*, "Support-set bottlenecks for video-text representation learning," 2021, arXiv: 2010.02824 [cs.CV].

[37] Jesús Andrés Portillo-Quintero, José Carlos Ortiz-Bayliss, and Hugo Terashima-Marin, "A straightforward framework for video retrieval using CLIP. 2021. arXiv: 2102.12443 [cs.CV].

[38] Alec Radford *et al.*, "Learning transferable visual models from natural language supervision," in *Image 2 ()*, T2.

[39] Alec Radford *et al.*, "Learning transferable visual models from natural language supervision," 2021, arXiv: 2103.00020 [cs.CV].

[40] Anna Rohrbach *et al.*, "Movie description," 2016, arXiv: 1605 . 03705 [cs.CV].

[41] Andrew Rouditchenko *et al.*, "AVLnet: learning audio-visual language representations from instructional videos," 2020, arXiv: 2006.09199 [cs.CV].

[42] Piyush Sharma *et al.*, "Conceptual captions: a cleaned, hypernymed, image alt-text dataset for automatic image captioning," 2018.

[43] Kihyuk Sohn, "Improved deep metric learning with multi-class N-pair loss objective," in *Advances in Neural Information Processing Systems*, Ed. by D. Lee et al. Vol. 29. Curran Associates, Inc., 2016.

[44] Xin Wang *et al.*, "VATEX: a large-scale, high-quality multilingual dataset for video-and-language research," 2020, arXiv: 1904.03493 [cs.CV].

[45] Jun Xu *et al.*, "MSR-VTT: a large video description dataset for bridging video and language," in *IEEE International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[46] Jianwei Yang, Yonatan Bisk, and Jianfeng Gao, "TACo: token-aware cascade contrastive learning for video-text alignment," 2021, arXiv: 2108. 09980 [cs.CV].

[47] Peter Young *et al.*, "From image descriptions to visual denotations: new similarity metrics for semantic inference over event descriptions," Cambridge, MA, 2014. doi: 10.1162/tacl_a_00166.

[48] Youngjae Yu, Jongseok Kim, and Gunhee Kim, "A joint sequence fusion model for video question answering and retrieval," 2018, arXiv: 1808. 02559 [cs.CV].

[49] Youngjae Yu *et al.*, "End-to-end concept word detection for video captioning, retrieval, and question answering," 2017, arXiv: 1610.02947 [cs.CV].

[50] Richard Zhang, "Making convolutional networks shift-invariant again," in *CoRR abs/1904.11486 (2019)*, arXiv: 1904.11486.

[51] Luowei Zhou, Chenliang Xu, and Jason J Corso, "Towards automatic learning of procedures from web instructional videos," in *AAAI Conference on Artificial Intelligence*, 2018, pp. 7590–7598.

Wasserstein Robust Reinforcement Learning

Haitham Bou-Ammar [1], Hang Ren [2,*], Mohammed Amin Abdullah [3,*], Vladimir Milenković [4,*], Rui Luo [5,*], Mingtian Zhang [5,*], Jun Wang [1,5]

[1] Huawei Noah's Ark
[2] Bloomberg LP
[3] Mozn.AI
[4] University of Cambridge
[5] University College London

Abstract

Despite their success, reinforcement learning (RL) algorithms tend to overfit to training environments, hampering their application in the real world. In this paper, we propose a robust RL algorithm that achieves significant robust performance on low- and high-dimensional control tasks. We formalize robust RL as a max-min game with a constraint derived from Wasserstein distance. Our efficient and scalable algorithm — called WR^2L — is designed to solve this game by applying a novel zero-order optimization method. We empirically demonstrate significant gains compared to standard and robust state-of-the-art algorithms on high-dimensional MuJoCo environments.

* Work done while at Huawei

1 Introduction

Reinforcement learning (RL) has become a standard tool for solving decision-making problems with feedback. Although there has been significant progress, RL algorithms often overfit to training environments and fail to generalize across even slight variations of transition dynamics [1, 2]. However, robustness to changes in transition dynamics is a crucial component for adaptive and safe RL in real-world environments.

To address the preceding problems, recent literature has proposed a plethora of algorithms for robust decision-making [3–5]. Most of these techniques borrow from game theory to analyze — typically in discrete state and action spaces — worst-case deviations of agents' policies and environments. For some examples, see [6–9] and the references therein. These methods have also been extended to linear function approximators [10] and deep neural networks [11], showing modest improvements in performance gain across a variety of disturbances such as action uncertainties and dynamics model variations.

In this paper, we propose a generic framework — one designed to handle both discrete and continuous state and action spaces — for robust RL. Our algorithm, termed *Wasserstein robust reinforcement learning* (WR^{2}L), judges any given policy against the worst-case dynamics among all candidate dynamics in a certain set in order to find the best policy. This set is essentially the average Wasserstein ball around the reference dynamics $\mathcal{P}_0$. The constraints make the problem well-defined, as searching over arbitrary dynamics can only result in system failure. The measure of performance is the standard RL objective, namely, the expected return. Both the policy and the dynamics are parameterized: the policy parameters θ_k may be the weights of a deep neural network, and the dynamics parameters ϕ_j may be the settings of a simulator or differential equation solver. The algorithm performs estimated descent steps in the ϕ space and — after convergence is complete or approaching completion — updates policy parameters in the θ space. Because ϕ_j may be high-dimensional, we adapt a zero-order sampling method based on [12] to estimate gradients. Furthermore, in order to define the constraint set that ϕ_j is bounded by, we generalize the method to estimate Hessians (Proposition 2). Although access to a simulator with parameterizable dynamics is required, it is important to note that the reference dynamics $\mathcal{P}_0$ needs not be known explicitly nor learned by our algorithm. Put another way, we are in the ''RL setting,'' not the "Markov decision process (MDP) setting" where the transition probability matrix is known **a priori**. The difference is made obvious, for example, in the fact that we cannot perform dynamics programming, and that the determination of a particular probability transition can only be estimated from sampling (not retrieved explicitly). As such, our algorithm is not model-based in the traditional sense of learning a model to perform planning.

We believe our contribution is useful and novel for two main reasons. First, our framing of the robust learning problem is in terms of dynamics uncertainty sets defined by Wasserstein distance. Although we are not the first to introduce Wasserstein distance into the context of MDPs (e.g., see [13, 14]), we believe our formulation is one of the first that can be applied to the demanding application space we desire — high-dimensional, continuous state and action space. Second, we believe our solution approach is both novel and effective (as evidenced by experiments in Section 5), and does not place a great demand on model or domain knowledge. Instead, it only needs access to a simulator or differentiable equation solver that allows for the parameterization of dynamics. Furthermore, it is not computationally demanding, especially because it does not build a model of the dynamics and because operations involving matrices are efficiently executable using the Jacobian-vector product facility of automatic differentiation engines.

2 Background

A Markov decision process (MDP)[1] is denoted by $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S} \subseteq \mathbb{R}^d$ denotes the state space, $\mathcal{A} \subseteq \mathbb{R}^n$ denotes the action space, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a state transition probability describing the system's dynamics, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function measuring the agent's performance, and $\gamma \in [0, 1)$ specifies the degree to which rewards are discounted over time.

At each time step t, the agent is in state $\boldsymbol{s}_t \in \mathcal{S}$ and must choose an action $\boldsymbol{a}_t \in \mathcal{A}$, transitioning itself to a new state $\boldsymbol{s}_{t+1} \sim \mathcal{P}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ and yielding a reward $\mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t)$. A policy $\pi : \mathcal{S} \times A \to [0, 1]$ is defined as a probability distribution over state-action pairs, where $\pi(\boldsymbol{a}_t|\boldsymbol{s}_t)$ represents the density of selecting action $\boldsymbol{a}_t$ in state $\boldsymbol{s}_t$. Upon subsequent interactions with the environment, the

[1] Please note that we present RL with continuous states and actions. This allows us to easily draw similarities to optimal control as detailed later. Extending these notions to discrete settings is relatively straightforward.

agent collects a trajectory $\boldsymbol{\tau}$ of state-action pairs. The goal is to determine an optimal policy $\pi^{\star}$ by solving:

$$\pi^{\star} = \arg\max_{\pi} \mathbb{E}_{\boldsymbol{\tau}\sim p_{\pi}(\boldsymbol{\tau})}\left[\mathcal{R}_{\text{Total}}(\boldsymbol{\tau})\right], \tag{1}$$

where $p_{\pi}(\boldsymbol{\tau})$ denotes the trajectory density function induced by $\mathcal{P}$ and π, and $\mathcal{R}_{\text{Total}}(\boldsymbol{\tau})$ is the return:

$$\mathcal{R}_{\text{Total}}(\boldsymbol{\tau}) = \sum_{t=0}^{T-1} \gamma^{t}\mathcal{R}(\boldsymbol{s}_t, \boldsymbol{a}_t).$$

We make use of **Wasserstein distance** to quantify variations from a reference transition density $\mathcal{P}_0(\cdot)$. Because this density is a probability distribution, one may consider other divergences, such as Kullback-Leibler (KL) or total variation (TV). We explain the main reasoning behind why we chose Wasserstein distance later, but here we point out a number of its desirable properties: First, it is symmetric ($W_p(\mu,\nu) = W_p(\nu,\mu)$, a property that KL lacks. Second, it is well-defined for measures with different supports (which KL also lacks). Indeed, Wasserstein distance is flexible in the forms of measures that can be compared: discrete, continuous, or a mixture. And third, it considers the underlying geometry of the space on which the distributions are defined, allowing valuable information to be encoded. This space is defined as follows: Let $\mathcal{X}$ be a metric space with metric $d(\cdot,\cdot)$, $\mathcal{C}(\mathcal{X})$ be the space of continuous functions on $\mathcal{X}$, and $\mathcal{M}(\mathcal{X})$ be the set of probability measures on $\mathcal{X}$. In addition, let $\mu,\nu \in \mathcal{M}(\mathcal{X})$, and $\mathbf{K}(\mu,\nu)$ be the set of couplings between μ,ν:

$$\begin{aligned}\mathbf{K}(\mu,\nu) :=&\{\kappa \in \mathcal{M}(\mathcal{X}\times\mathcal{X})\,;\,\forall(A,B)\subset\mathcal{X}\times\mathcal{X},\\ &\kappa(A\times\mathcal{X}) = \mu(A), \kappa(\mathcal{X}\times B) = \nu(B)\}\end{aligned} \tag{2}$$

That is, the set of joint distributions $\kappa \in \mathcal{M}(\mathcal{X}\times\mathcal{X})$ whose marginals agree with μ and ν. Given a metric (serving as a cost function) $d(\cdot,\cdot)$ for $\mathcal{X}$, the p'th Wasserstein distance $W_p(\mu,\nu)$ for $p\geq 1$ between μ and ν is defined as:

$$W_p(\mu,\nu) := \left(\min_{\kappa\in\mathbf{K}(\mu,\nu)} \int_{\mathcal{X}\times\mathcal{Y}} d(x,y)^p d\kappa(x,y)\right)^{1/p} \tag{3}$$

In this paper, and mostly for computational convenience, we use $p = 2$, though other values of p are applicable.

3 WR²L

The desirable properties of Wasserstein distance aside, the main reasoning behind why we chose it is as follows: According to the preceding definition, constraining the possible dynamics to be within an ϵ-Wasserstein ball of the reference dynamics $\mathcal{P}_0(\cdot)$ means constraining it in a certain way. Wasserstein distance has the form of mass × distance. If this quantity is constrained to be less than a constant ϵ, the distance is small if the mass is large, and conversely, the mass is small if the distance is large. Intuitively, when modelling the dynamics of a system, it may be reasonable to concede that there could be a systemic error — or bias — in the model. However, such bias should not be too large. It is also reasonable to suppose that the behavior of the system may occasionally deviate from the model, but this should be a low-probability event. A model that is frequently wrong by a large amount is of no use. In a sense, the Wasserstein ball formalizes these assumptions.

3.1 Problem Definition: Robust Objectives and Constraints

Given the continuous nature of the state and action spaces considered in this paper, we utilize deep neural networks to parameterize policies. We write these policies as $\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t)$, where $\boldsymbol{\theta}\in\mathbb{R}^{d_1}$ is a set of tunable hyperparameters to optimize. For instance, these policies can correspond to multilayer perceptrons for MuJoCo environments, or to convolutional neural networks in case of high-dimensional states depicted as images. Ultimately, the exact details of policies are application-dependent; as such, we discuss those details in the relevant experiment sections.

In principle, one can similarly parameterize dynamics models using deep neural networks (e.g., LSTM-type models) to provide one or more action-conditioned future state predictions. Though appealing, **our initial experiments with this approach gave rise to transition models that lack valid physical meaning**. For example, CartPole ended up involving transitions that alter angles without changing angular velocities. These effects became more apparent in high-dimensional settings where the number of potential minimizers increases significantly. It is worth noting that we are not the first to realize such effects when attempting to model physics-based dynamics using deep neural networks. Authors in [15] remedy these problems by introducing Lagrangian mechanics to deep neural networks, while others [16, 17] argue for the need to directly model dynamics given by differential equation structures.

Though incorporating physics-based priors into deep neural networks is an important and challenging task that holds the promise of scaling model-based RL for efficient solvers, we study an alternative direction in this paper. We focus on perturbing differential equation solvers and simulators with respect to the dynamics specification parameters $\boldsymbol{\phi}\in\mathbb{R}^{d_2}$, which would not only reduce the dimensions of parameter spaces representing transition models, but also guarantee valid dynamics due to the nature of the simulator. In tackling some of the preceding problems, such a direction involves a new set of challenges related to computing gradients and Hessians of black-box solvers. In Section 4, we develop an efficient and scalable zero-order optimization method for valid and accurate model updates.

Borrowing from robust optimal control, we define robust RL as an algorithm that learns best-case policies under worst-case transitions:

$$\max_{\boldsymbol{\theta}} \left[\min_{\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{\tau} \sim p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau})} \left[\mathcal{R}_{\text{total}}(\boldsymbol{\tau}) \right] \right], \tag{4}$$

where $p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau})$ is a trajectory density function parameterized by both policies $\boldsymbol{\theta}$ and transition models $\boldsymbol{\phi}$:

$$p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau}) =$$
$$\mu_0(\boldsymbol{s}_0)\pi(\boldsymbol{a}_0|\boldsymbol{s}_0) \prod_{t=1}^{T-1} \underbrace{\mathcal{P}_{\boldsymbol{\phi}}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)}_{\text{specs vector and diff. solver}} \quad \underbrace{\pi_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t)}_{\text{deep network}}$$

where $\mu_0(\cdot)$ is the initial state distribution. Although inspired by robust optimal control, our formulation is more generic because it allows for parameterized classes of transition models without incorporating additional restrictions on the structure or the scope by which variations are executed[2].

Due to the arbitrary class of parameterized transitions, the problem in Equation 4 is ill-defined. To ensure well-behaved optimization objectives, we introduce constraints to bound search spaces and ensure convergence to feasible transition models. For a valid constraint set, our method assumes access to samples from a **reference dynamics model** $\mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})$, and bounds learned transitions in an ϵ-Wasserstein ball around $\mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})$, i.e., the set is defined as:

$$\begin{aligned} &\mathcal{W}_\epsilon\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot), \mathcal{P}_0(\cdot)\right) = \{\mathcal{P}_{\boldsymbol{\phi}}(\cdot) : \\ &\mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s}, \boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})\right) \le \epsilon, \forall(\boldsymbol{s}, \boldsymbol{a}) \in \mathcal{S} \times \mathcal{A}\, \} \end{aligned} \tag{5}$$

where $\epsilon \in \mathbb{R}_+$ is a hyperparameter used to specify the "degree of robustness" like maximum norm bounds in robust optimal control. Although we have access to samples from a reference simulator, our setting is by no means restricted to model-based RL in an MDP setting. That is, our algorithm operates successfully with only traces from $\mathcal{P}_0$ accompanied with its specification parameters (e.g., pole length and torso mass) — it does not require full model learners and is therefore more flexible. Note that Equation 5 introduces an infinite number of constraints when considering continuous state and action spaces. Consequently, we consider a relaxation of **average** Wasserstein distance bounded by a hyperparameter ϵ:

$$\begin{aligned} &\hat{\mathcal{W}}_\epsilon^{(\text{average})}\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot), \mathcal{P}_0(\cdot)\right) = \{\mathcal{P}_{\boldsymbol{\phi}}(\cdot) : \\ &\int_{(\boldsymbol{s},\boldsymbol{a})} \mathcal{P}(\boldsymbol{s}, \boldsymbol{a}) \mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s}, \boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})\right) d(\boldsymbol{s}, \boldsymbol{a}) \le \epsilon\, \} \\ &= \left\{\mathcal{P}_{\boldsymbol{\phi}}(\cdot) : \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})}\left[\mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s}, \boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})\right)\right] \le \epsilon\right\}. \end{aligned} \tag{6}$$

The sampling $(\boldsymbol{s}, \boldsymbol{a})$ in the expectation is done as follows: We sample trajectories using reference dynamics $\mathcal{P}_0$ and a policy π that chooses actions uniformly at random (**UAR**). Then $(\boldsymbol{s}, \boldsymbol{a})$ pairs are sampled **UAR** from those collected trajectories. For a given pair $(\boldsymbol{s}, \boldsymbol{a})$, $\mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s}, \boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})\right)$ is approximated through the empirical distribution: we use the state that followed $(\boldsymbol{s}, \boldsymbol{a})$ in the collected trajectories as a data point. Estimating Wasserstein distance using empirical data is standard practice (e.g., see [18]). One approach that worked well in our experiments was to assume that the dynamics is given by deterministic functions plus Gaussian noise with diagonal covariance matrices. This makes estimation easier in high dimensions because sampling in each dimension is independent of others, and the total number of samples needed is a constant factor of the number of dimensions. Gaussian distributions also have closed-form expressions for Wasserstein distance, given in terms of mean and covariance.

As such, we arrive at WR²L's optimization problem allowing for the best policies to be found under worst-case yet bounded transition models:

WR²L Objective:

$$\begin{aligned} &\max_{\boldsymbol{\theta}} \left[\min_{\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{\tau} \sim p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau})}\left[\mathcal{R}_{\text{total}}(\boldsymbol{\tau})\right]\right] \\ &\text{s.t.}\quad \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})}\left[\mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s}, \boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})\right)\right] \le \epsilon \end{aligned} \tag{7}$$

3.2 Solution Methodology

Our solution alternates between updates of $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, keeping one fixed while updating the other. When dynamics parameters $\boldsymbol{\phi}$ are fixed, policy parameters $\boldsymbol{\theta}$ can be updated by solving $\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{\tau} \sim p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau})}\left[\mathcal{R}_{\text{total}}(\boldsymbol{\tau})\right]$, which is the formulation of a standard RL problem. Consequently, it is easy to adapt any policy search method for updating policies under fixed dynamics models. As described later in Section 4, we make use of proximal policy optimization (PPO) [19]. When updating $\boldsymbol{\phi}$ with the given $\boldsymbol{\theta}$, we must respect the Wasserstein constraint. Unfortunately, even with the simplification introduced in Section 3.1, the constraint is still difficult to compute. We therefore approximate the constraint in (7) by its Taylor expansion up to second order. That is, defining $W(\boldsymbol{\phi}) := \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})}\left[\mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s}, \boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s}, \boldsymbol{a})\right)\right]$,

[2] Ultimately, allowed perturbations are constrained by the hypothesis space. Even so, our model is more generic compared to robust optimal control that assumes additive, multiplicative, or other forms of disturbances.

the constraint can be approximated around ϕ_0 by a second-order Taylor expansion as:

$$W(\phi) \approx W(\phi_0) + \nabla_{\phi} W(\phi_0)^{\mathsf{T}} (\phi - \phi_0) + \frac{1}{2} (\phi - \phi_0)^{\mathsf{T}} \nabla^2_{\phi} W(\phi_0) (\phi - \phi_0).$$

Because $W(\phi_0) = 0$ (zero distance between a density and itself) and $\nabla_{\phi} W(\phi_0) = 0$ (ϕ_0 minimizes $W(\phi)$), we can simplify the Hessian approximation as follows: $W(\phi) \approx \frac{1}{2}(\phi - \phi_0)^{\mathsf{T}} \nabla^2_{\phi} W(\phi_0)(\phi - \phi_0)$. Given $\boldsymbol{\theta}$, the inner minimization becomes

$$\min_{\phi} \mathbb{E}_{\boldsymbol{\tau} \sim p^{\phi}_{\boldsymbol{\theta}}(\boldsymbol{\tau})} [\mathcal{R}_{\text{total}}(\boldsymbol{\tau})] \quad \text{s.t.} \quad \frac{1}{2}(\phi - \phi_0)^{\mathsf{T}} \boldsymbol{H}_0 (\phi - \phi_0) \le \epsilon, \tag{8}$$

where $\boldsymbol{H}_0 = \nabla^2_{\phi} \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})} \left[\mathcal{W}_2^2 \left(\mathcal{P}_{\phi}(\cdot|\boldsymbol{s},\boldsymbol{a}), \mathcal{P}_0(\cdot|\boldsymbol{s},\boldsymbol{a}) \right) \right] \Big|_{\phi = \phi_0}$ is the Hessian of the expected squared 2-Wasserstein distance evaluated at ϕ_0. Optimization problems with quadratic constraints can be efficiently solved using interior-point methods. To do so, one typically approximates the loss with a first-order expansion and determines a closed-form solution. Consider a pair of parameters $\boldsymbol{\theta}^{[k]}$ and $\phi^{[j]}$ (which correspond to parameters of the k'th outer loop and the j'th inner loop, respectively, in the algorithm we present). To find $\phi^{[j+1]}$, we solve:

$$\min_{\phi} \nabla_{\phi} \mathbb{E}_{\boldsymbol{\tau} \sim p^{\phi}_{\boldsymbol{\theta}}(\boldsymbol{\tau})} [\mathcal{R}_{\text{total}}(\boldsymbol{\tau})] \Big|^{\mathsf{T}}_{\boldsymbol{\theta}^{[k]}, \phi^{[j]}} (\phi - \phi^{[j]})$$

$$\text{s.t.} \quad \frac{1}{2}(\phi - \phi_0)^{\mathsf{T}} \boldsymbol{H}_0 (\phi - \phi_0) \le \epsilon.$$

A minimizer to the above equation can be derived in a closed form as:

$$\phi^{[j+1]} = \phi_0 - \sqrt{\frac{2\epsilon}{\boldsymbol{g}^{[k,j]\mathsf{T}} \boldsymbol{H}_0^{-1} \boldsymbol{g}^{[k,j]}}} \boldsymbol{H}_0^{-1} \boldsymbol{g}^{[k,j]}, \tag{9}$$

with $\boldsymbol{g}^{[k,j]}$ denoting the gradient[3] evaluated at $\boldsymbol{\theta}^{[k]}$ and $\phi^{[j]}$, specifically, $\boldsymbol{g}^{[k,j]} = \nabla_{\phi} \mathbb{E}_{\boldsymbol{\tau} \sim p^{\phi}_{\boldsymbol{\theta}}(\boldsymbol{\tau})} \mathbb{E} [\mathcal{R}_{\text{total}}(\boldsymbol{\tau})] |_{\boldsymbol{\theta}^{[k]}, \phi^{[j]}}$.

Generic Algorithm: Having described the two main steps needed for updating policies and models, we now summarize these findings in the pseudocode in Algorithm 1. As the Hessian[4] of the Wasserstein distance is evaluated based on reference dynamics and any policy π, we pass it (together with ϵ and ϕ_0) as an input. Then Algorithm 1 operates in a descent-ascent fashion in two main phases. In the first phase (lines 5–10), dynamics parameters are updated using (9), while ensuring that learning rates abide by step-size conditions (we used Wolfe conditions [21]). The second phase (line 11) utilizes any state-of-the-art RL method to adapt policy parameters that generate $\boldsymbol{\theta}^{[k+1]}$. Regarding the termination condition for the inner loop, we leave this as a decision for the user. It could be, for example, a large finite time-out, or the norm of the gradient $\boldsymbol{g}^{[k,j]}$ being below a threshold, or whichever occurs first.

[3] **Remark:** Superficially, this looks similar to an approximation made in trust region policy optimization (TRPO) [20]. However, the latter aims to optimize the **policy parameter** rather than dynamics. Furthermore, the constraint is based on KL divergence rather than Wasserstein distance.

[4] Note that our algorithm does not need to store the Hessian matrix; we require only Hessian-vector products. These products can be easily computed using computational graphs without having access to the full matrix.

Algorithm 1 WR²L

```
1: Inputs: Wasserstein distance Hessian, $\boldsymbol{H}_0$ evaluated at $\phi_0$
   under any policy $\pi$, radius of the Wasserstein ball $\epsilon$, and
   the reference simulator specification parameters $\phi_0$
2: Initialize $\phi^{[0]}$ with $\phi_0$ and policy parameters $\boldsymbol{\theta}^{[0]}$ arbitrarily
3: for $k = 0, 1, \dots$ do
4:     $\boldsymbol{x}^{[0]} \leftarrow \phi^{[0]}$, and $j \leftarrow 0$
5:     Phase I: Update model parameters while fixing the
       policy:
6:     while termination condition not met do
7:         Compute the descent direction for model parameters
           as given by Equation 9:
           $\boldsymbol{p}^{[j]} \leftarrow \phi_0 - \sqrt{\frac{2\epsilon}{\boldsymbol{g}^{[k,j]\mathsf{T}} \boldsymbol{H}_0^{-1} \boldsymbol{g}^{[k,j]}}} \boldsymbol{H}_0^{-1} \boldsymbol{g}^{[k,j]} - \boldsymbol{x}^{[j]}$
8:         Update candidate solutions while satisfying step-size
           conditions (see discussion below) on the learning rate
           $\alpha$: $\boldsymbol{x}^{[j+1]} \leftarrow \boldsymbol{x}^{[j]} + \alpha \boldsymbol{p}^{[j]}$ and $j \leftarrow j + 1$
9:     end while
10:    Perform model update setting $\phi^{[k+1]} \leftarrow \boldsymbol{x}^{[j]}$
11:    Phase II: Update the policy with the new model
       parameters:
12:    Use any standard RL algorithm for ascending in the
       gradient direction, for example,
       $\boldsymbol{\theta}^{[k+1]} \leftarrow \boldsymbol{\theta}^{[k]} + \beta^{[k]} \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{\tau} \sim p^{\phi}_{\boldsymbol{\theta}}(\boldsymbol{\tau})} [\mathcal{R}_{\text{total}}(\boldsymbol{\tau})] |_{\boldsymbol{\theta}^{[k]}, \phi^{[k+1]}}$,
       with $\beta^{[k]}$ being the learning rate.
13: end for
```

4 Zero-Order WR²L

Consider a simulator (or differential equation solver) $\mathbb{S}_{\phi}$ for which the dynamics are parameterized by a real vector ϕ, and for which we can execute steps of a trajectory (i.e., the simulator takes an action $\boldsymbol{a}$ as input and returns a successor state and reward). To generate novel physics-grounded transitions, one can simply alter ϕ and execute the instruction in $\mathbb{S}_{\phi}$ from some state $\boldsymbol{s} \in \mathcal{S}$, while applying an action $\boldsymbol{a} \in \mathcal{A}$. This not only ensures valid (under mechanics) transitions, but also promises scalability because specification parameters typically reside in lower-dimensional spaces compared to the number of tunable weights when using deep neural networks as transition models.

Recalling the update rule in Phase I of Algorithm 1, we realize the need for estimating the gradient of the loss function with respect to the vector specifying the dynamics of the environment, specifically, $\boldsymbol{g}^{[k,j]} = \nabla_{\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{\tau}\sim p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau})}\left[\mathcal{R}_{\text{total}}(\boldsymbol{\tau})\right]\Big|_{\boldsymbol{\theta}^{[k]},\boldsymbol{\phi}^{[j]}}$ at each iteration of the inner loop j. Handling simulators as black-box models, we estimate the gradients as follows (which is relatively easy to prove):

Proposition 1 (Zero-Order Gradient Estimate). *For a fixed $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, the gradient can be computed as:*

$$\nabla_{\boldsymbol{\phi}} \mathbb{E}_{\boldsymbol{\tau}\sim p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}}(\boldsymbol{\tau})}\left[\mathcal{R}_{\text{total}}(\boldsymbol{\tau})\right] = \frac{1}{\sigma^2}\mathbb{E}_{\boldsymbol{\xi}\sim\mathcal{N}(\mathbf{0},\sigma^2\boldsymbol{I})}\left[\boldsymbol{\xi}\int_{\boldsymbol{\tau}} p_{\boldsymbol{\theta}}^{\boldsymbol{\phi}+\boldsymbol{\xi}}(\boldsymbol{\tau})\mathcal{R}_{\text{total}}(\boldsymbol{\tau})d\boldsymbol{\tau}\right].$$

We can extend the above for Hessians with the following: **Proposition 2** (Zero-Order Hessian Estimate). *The Hessian of the Wasserstein distance around $\boldsymbol{\phi}_0$ can be estimated based on function evaluations. Recalling that $\boldsymbol{H}_0 = \nabla^2_{\boldsymbol{\phi}} \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})\sim\pi(\cdot)\rho_{\pi}^{\boldsymbol{\phi}_0}(\cdot)}\left[\mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s},\boldsymbol{a}),\mathcal{P}_0(\cdot|\boldsymbol{s},\boldsymbol{a})\right)\right]\Big|_{\boldsymbol{\phi}=\boldsymbol{\phi}_0}$, and defining $\mathcal{W}_{(\boldsymbol{s},\boldsymbol{a})}(\boldsymbol{\phi}) := \mathcal{W}_2^2\left(\mathcal{P}_{\boldsymbol{\phi}}(\cdot|\boldsymbol{s},\boldsymbol{a}),\mathcal{P}_0(\cdot|\boldsymbol{s},\boldsymbol{a})\right)$, we prove:*

$$\boldsymbol{H}_0 = \frac{1}{\sigma^2}\mathbb{E}_{\boldsymbol{\xi}\sim\mathcal{N}(\mathbf{0},\sigma^2\boldsymbol{I})} \left[\frac{1}{\sigma^2}\boldsymbol{\xi}\left(\mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})\sim\pi(\cdot)\rho_{\pi}^{\boldsymbol{\phi}_0}(\cdot)}\left[\mathcal{W}_{(\boldsymbol{s},\boldsymbol{a})}\left(\boldsymbol{\phi}_0+\boldsymbol{\xi}\right)\right]\right)\boldsymbol{\xi}^{\mathsf{T}} - \mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})\sim\pi(\cdot)\rho_{\pi}^{\boldsymbol{\phi}_0}(\cdot)}\left[\mathcal{W}_{(\boldsymbol{s},\boldsymbol{a})}(\boldsymbol{\phi}_0+\boldsymbol{\xi})\right]\boldsymbol{I}\right].$$

We reiterate that the Hessian estimation needs to be made once only — before executing the instructions in Algorithm 1 (i.e., $\boldsymbol{H}_0$ is passed as an input). With the above, gradients and Hessian estimates can be simply based on simulator value evaluations while perturbing $\boldsymbol{\phi}$ and $\boldsymbol{\phi}_0$. This requires evaluation of $\mathbb{E}_{(\boldsymbol{s},\boldsymbol{a})\sim\pi(\cdot)\rho_{\pi}^{\boldsymbol{\phi}_0}(\cdot)}\left[\mathcal{W}_{(\boldsymbol{s},\boldsymbol{a})}(\boldsymbol{\phi}_0)\right]$ under random $\boldsymbol{\xi}$ perturbations sampled from $\mathcal{N}(\mathbf{0},\sigma^2\boldsymbol{I})$, which can be done through empirical estimations. That is, approximating μ by $\mu_n = \frac{1}{n}\sum_{i=1}^n \delta_{\boldsymbol{x}_i}$ where $\boldsymbol{x}_i$ is i.i.d sampled from μ, and similarly for ν, we have[5] $\mathcal{W}_2(\mu,\nu)\approx\mathcal{W}_2(\mu_n,\nu_n)$.

5 Experiments & Results

We evaluate WR^{2}L on a variety of continuous control benchmarks from the MuJoCo environment. Dynamics in our benchmarks was parameterized by variables defining physical behavior (e.g., density of the robot's torso and friction of the ground). We consider both low- and high-dimensional dynamics and demonstrate that our algorithm outperforms state-of-the-art algorithms from both standard and robust RL. We are primarily interested in policy generalization across environments with varying dynamics, which we measure with average test returns on novel systems. The comparison against standard RL algorithms allows us to understand whether the lack of robustness is a critical challenge for sequential decision making. At the same time, the comparison against robust RL algorithms tests if we outperform state-of-the-art algorithms that consider a similar setting to ours. From standard RL algorithms, we compare against PPO [19], and TRPO [20]; an algorithm based on natural actor-critic [23, 24]. From robust RL algorithms, we demonstrate how WR^{2}L favors against robust adversarial reinforcement learning (RARL) [4], and action-perturbed Markov decision processes (PR-MDP) proposed in [5]. It is worth noting that we attempted to include deep deterministic policy gradients (DDPG) [25] in our comparisons. However, results including DDPG were omitted as DDPG failed to show any significant robustness performance even on relatively simple systems, such as the inverted pendulum. During initial trials, we also performed experiments to parameterize models using deep neural networks. Results demonstrated that these models, though minimizing training data error, failed to provide valid physics-grounded dynamics. For instance, we arrived at inverted pendulum models that varied pole angles without exerting any angular speeds. This problem became even more apparent in high-dimensional systems (e.g., Hopper and Walker) due to the increased number of possible minima. As such, results presented in this section make use of our zero-order method that can be regarded as a scalable alternative for robust solutions.

5.1 MuJoCo Benchmarks

We evaluate our method both in low- and **high-dimensional** MuJoCo tasks [26]. We consider a variety of benchmarks including CartPole, Hopper, and Walker2D, all of which require direct joint-torque control. Keeping with the generality of our method, we utilize these benchmarks as-is with no additional alterations. Specifically, we use the exact setting of these benchmarks as that shipped with OpenAI gym without any reward shaping, state-space augmentation, feature extraction, or any other similar modifications.

Results for one-dimensional parameter variations show that WR^{2}L outperforms both robust and non-robust algorithms when one-dimensional simulator variations are considered.

[5] If the dynamics is assumed to be Gaussian, a similar procedure can be followed or a closed form can be used — see [22].

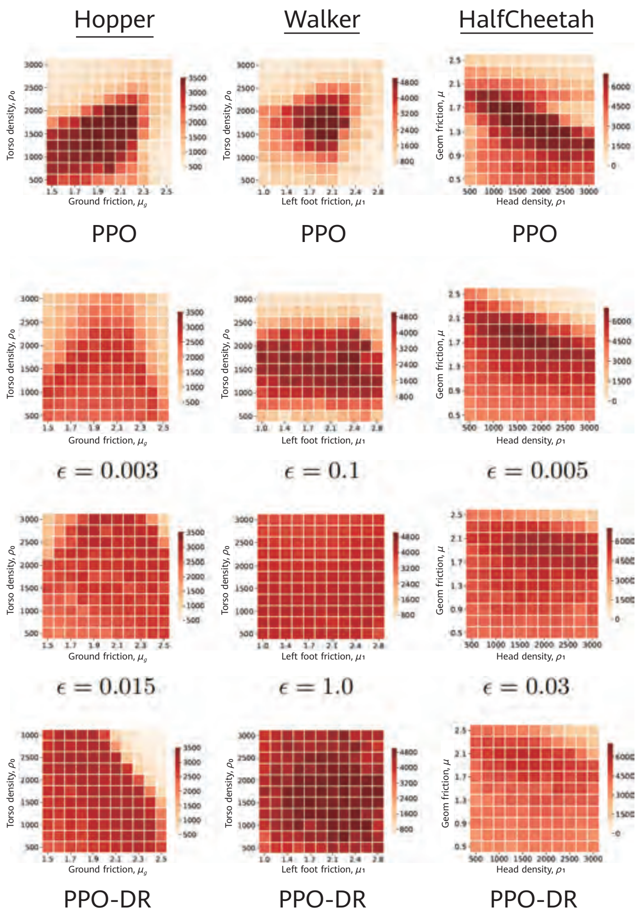


Figure 1 Two-dimensional dynamics variations. Numbers in the graded scale are scores that the tasks can generate. Darker is better.

Figure 1 shows results for dynamics variations along two dimensions. Here again, our method demonstrates considerable robustness. The fourth row, "PPO-DR", refers to experiments where PPO is trained on dynamics sampled **UAR** from the range of parameters displayed. For example, PPO-DR in the Hopper column is trained with pairs of <torso density, ground friction> in the range [500, 3000] × [1.5, 2.5]. It displays more robustness than when trained on just the reference dynamics. However, as can be seen from Figure 2, our method performs better in high dimensions, which is the main strength of our algorithm.

Results with High-Dimensional Model Variations: Although the preceding results demonstrate robustness, an argument against a min-max objective can be made especially when only low-dimensional changes in the simulator are considered. Specifically, one can argue the need for such an objective as opposed to simply sampling a set of systems and determining policies performing well on average similar to the approach proposed in [27].

A counterargument is that a gradient-based optimization scheme is more efficient than a sampling-based one when high-dimensional changes are considered. In other words, a sampling procedure is hardly applicable when more than a few parameters are altered, while WR²L can remain suitable. To assess these claims, we conducted two additional experiments on the Hopper and HalfCheetah benchmarks. In the first, we trained robustly while changing friction and torso densities, and tested on 1000 systems generated by varying all 11 dimensional parameters of the Hopper benchmark, and 21 dimensional parameters of the HalfCheetah benchmark.

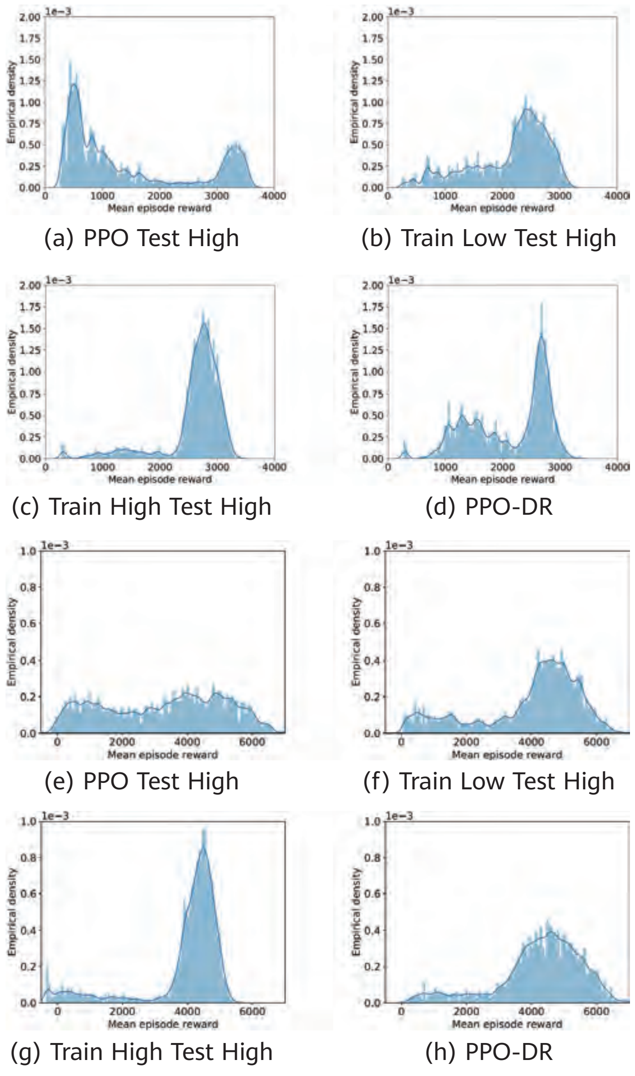


Figure 2 High-dimensional dynamics variations for Hopper (a–d) and HalfCheetah (e–h)

The histograms in Figures 2b and f demonstrate that the empirical densities of the average test returns are mostly centered on 3000 for Hopper, and on 4500 for HalfCheetah. This is an improvement over PPO trained on reference dynamics (Figures 2a and e) with return masses mostly accumulated at around 1000 in the case of the Hopper benchmark and almost equally distributed when considering HalfCheetah. Such improvements, however, can be an artifact of the careful choice of the low-dimensional degrees of freedom allowed to be modified during Phase I of Algorithm 1. For further insights, Figures 2c and g demonstrate the effectiveness of our method trained and tested while allowing to tune all 11 dimensional parameters of the Hopper benchmark, and 21 dimensional parameters of the HalfCheetah benchmark. Indeed, our results comply with those of the previous experiment, indicating that most of the test returns' mass remains around 3000 for Hopper, and improves to accumulate around 4500 for HalfCheetah. Interestingly, our algorithm can acquire higher returns on all systems because it is allowed to alter all parameters defining the simulators. As such, we conclude that WR^2L outperforms others when high-dimensional simulator variations are considered. In Figures 2d and h, we see the results for PPO trained with dynamics sampled **UAR** from the Wasserstein constraint set. Although this training method works well in the two-dimensional variation case, it does not scale well to high dimensions. Our method achieves better results.

6 Related Work

Work on robust MDPs (e.g., [8, 7, 28–30]), while valuable in its own right, is not sufficient for the RL setting due to the need in RL to give efficient solutions for large state and action spaces, and the fact that the dynamics is not known **a priori**. Closer to our problem space is a number of papers that do not assume known dynamics (e.g., [31–34]). [32] extended most conventional RL algorithms, including Q-learning, SARSA, and TD-learning, to their robust versions. Convergence proofs of two function approximations for large-scale MDPs were also conducted in [32]. However, two difficulties arise in robust RL. The first one inherits from the robust MDP framework in which the assumption of rectangular ambiguity sets[6] (i.e., dynamics) results in conservative policies. Second, further study is required on how to estimate the ambiguity sets for large-scale MDPs in a model-free setting [30, 33]. In our work, the dynamics is parameterized and the corresponding ambiguity set is non-rectangular. The ambiguity set is mapped to a subset in the model parameter space, which is assumed to a l_2-ball around a nominal parameter. Proposition 2 gives a method to estimate the ambiguity set through sampling. Unlike most value-based algorithms discussed here, we exploit a policy gradient approach that generalizes to complex systems with continuous state and action spaces. There are other robust RL frameworks that consider robustness under adversarial action noises: Pinto et al. [4] and Tessler et al. [5]. We tested against both of these algorithms and found they lacked performance even in the one-dimensional case. Rajeswaran et al. [27] approaches the robustness problem by performing alternating optimization for environment and policy parameters, choosing the dynamics sampled from a probability distribution over the parameter space, and using a fixed fraction of the worst-performing trajectories to update the policy. Despite some resemblance to our algorithm, there are crucial differences. First, it does not utilize Wasserstein distance. Second, our algorithm takes **descent steps** in the ϕ space. These differences are important when the dynamics parameters sit in a high-dimensional space. This is because, in that case, optimization-from-sampling could demand a considerable number of samples. In any case, our experiments demonstrate our algorithm performs well even in these high dimensions. Unfortunately, we were unable to find the code for the Rajeswaran et al. paper [27], and did not attempt to implement it ourselves.

[6] This means the choice of the transition probability at any (s, a) pair is completely independent on choices made at other pairs.

In [14], a non-stationary MDP is considered, where the dynamics can change from one time step to another. It is assumed the dynamics is known at each time step but not how it will change. It is also assumed that the dynamics variation is bounded, specifically, the Wasserstein distance between dynamics at time t and t' is Lipschitz. [14] approaches the problem by treating nature as an adversary and implementing a min-max algorithm. The resulting algorithm, known as *Risk Averse Tree Search*, is — as the name implies — a tree search algorithm. The algorithm was tested on a small grid world, and does not appear to be readily extendable to the continuous state and action spaces our algorithm addresses. To summarize, our paper uses Wasserstein distance for quantifying variations in possible dynamics, in common with [14], but is suited to applying deep neural networks for continuous state and action spaces. Our algorithm also does not require full dynamics; instead, it requires mere parameterizable dynamics. It competes well with the preceding papers, and operates well for high-dimensional problems, as evidenced by the experiments.

Domain Randomization: Domain randomization is an important technique to bridge the gap between the simulator and the real-world system [35, 11]. The objective has a max-expectation form in which the policy performance is evaluated over a distribution of simulator parameters. This distribution is manually designed at first and can then be adapted [27, 36, 37]. Our experimental setup is similar to that in domain randomization, which uses a parameterized simulator. That said, our method is distinguishable from two perspectives. First, we follow the max-min formula in the robust RL framework and take gradient steps to search the worst-case scenario. However, the max-expectation objective from domain randomization requires sampling simulator parameters from a distribution. This means that the objective suffers from the curse of dimensionality when applied to complex systems with many parameters (as shown in the empirical demonstration in the experiment section). Second, the ambiguity set (the support of the parameter distribution) in domain randomization is usually handcrafted, requiring additional expert knowledge. We learn the ambiguity set from data and consider it as a constraint.

7 Conclusion & Future Work

In this paper, we proposed a robust RL algorithm capable of outperforming others in terms of test returns on unseen dynamics. The algorithm makes use of Wasserstein constraints for policy generalization across varying domains, and considers a zero-order method for scalable solutions. Empirically, we demonstrated superior performance against state-of-the-art algorithms from both standard and robust RL on low- and high-dimensional MuJoCo environments. In future work, we aim to consider robustness in terms of other components of MDPs, for example, state representations, reward functions, and others. Furthermore, we will implement WR^{2}L on real hardware, considering sim-to-real experiments.

References

[1] Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song, "Assessing generalization in deep reinforcement learning," *arXiv prerint arXiv:1810.12282*, 2018.

[2] Chenyang Zhao, Olivier Sigaud, Freek Stulp, and Timothy M. Hospedales, "Investigating generalisation in continuous deep reinforcement learning," *arXiv preprint arXiv:1902.07015*, 2019.

[3] Jun Morimoto and Kenji Doya, "Robust reinforcement learning," *Neural Comput.*, 17(2):335–359, Feb 2005.

[4] Lerrel Pinto, James Davidson, Rahul Sukthankar, and Abhinav Gupta, "Robust adversarial reinforcement learning," in *Proceedings of International Conference on Machine Learning*, pages 2817–2826, Sydney, Australia, 06–11 Aug 2017.

[5] Chen Tessler, Yonathan Efroni, and Shie Mannor, "Action robust reinforcement learning and applications in continuous control," in *Proceedings of International Conference on Machine Learning*, pages 6215–6224, Long Beach, California, USA, 09–15 Jun 2019.

[6] Thomas Sargent and Lars Hansen, "Robust control and model uncertainty," *American Economic Review*, 91(2):60–66, 2001.

[7] Arnab Nilim and Laurent El Ghaoui, "Robust control of Markov decision processes with uncertain transition matrices," *Operations Research*, 53(5):780–798, 2005.

[8] Garud N Iyengar, "Robust dynamic programming," *Mathematics of Operations Research*, 30(2):257–280, 2005.

[9] Hongseok Namkoong and John C. Duchi, "Stochastic gradient methods for distributionally robust optimization with F-divergences," in *Advances in Neural Information Processing Systems*, pages 2216–2224, Barcelona, Spain, 2016.

[10] Yinlam Chow, Aviv Tamar, Shie Mannor, and

Marco Pavone, "Risk-sensitive and robust decision-making: A CVaR optimization approach," in *Advances in Neural Information Processing Systems*, pages 1522–1530. 2015.

[11] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *International Conference on Robotics and Automation*, pages 1–8, Brisbane, Australia, 21–25 May 2018. IEEE.

[12] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017.

[13] Insoon Yang, "A convex optimization approach to distributionally robust Markov decision processes with Wasserstein distance," *IEEE Control Syst Letters*, 1(1):164–169, 2017.

[14] Erwan Lecarpentier and Emmanuel Rachelson, "Nonstationary Markov decision processes, a worst-case approach using model-based reinforcement learning," in *Advances in Neural Information Processing Systems*, pages 7214–7223, 2019.

[15] Michael Lutter, Christian Ritter, and Jan Peters, "Deep Lagrangian networks: Using physics as model prior for deep learning," *arXiv preprint arXiv:1907.04490*, 2019.

[16] Torsten Koller, Felix Berkenkamp, Matteo Turchetta, and Andreas Krause, "Learning-based model predictive control for safe exploration and reinforcement learning," *arXiv preprint arXiv:1803.08287*, 2018.

[17] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, pages 6571–6583. 2018.

[18] Gabriel Peyré, Marco Cuturi, et al, "Computational optimal transport," *Foundations and Trends® in Machine Learning*, 11(5-6):355–607, 2019.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[20] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel, "Trust region policy optimization," in *Proceedings of International Conference on Machine Learning*, pages 1889–1897, Lille, France, 07–09 Jul 2015.

[21] Philip Wolfe, "Convergence conditions for ascent methods," *SIAM review*, 11(2):226–235, 1969.

[22] Asuka Takatsu, "Wasserstein geometry of Gaussian measures," *Osaka Journal of Mathematics*, 48(4):1005–1026, 2011.

[23] Jan Peters and Stefan Schaal, "Natural actor-critic," *Neurocomput.*, 71(7–9):1180–1190, Mar 2008.

[24] Joni Pajarinen, Hong Linh Thai, Riad Akrour, Jan Peters, and Gerhard Neumann, "Compatible natural gradient policy search," *arXiv preprint arXiv:1902.02823*, 2019.

[25] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of International Conference on Machine Learning*, pages I–387–I–395, 2014.

[26] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[27] Aravind Rajeswaran, Sarvjeet Ghotra, Balaraman Ravindran, and Sergey Levine, "Epopt: Learning robust neural network policies using model ensembles," *ICLR*, 2017.

[28] Wolfram Wiesemann, Daniel Kuhn, and Berç Rustem, "Robust Markov decision processes," *Mathematics of Operations Research*, 38(1):153–183, 2013.

[29] Andrea Tirinzoni, Marek Petrik, Xiangli Chen, and Brian Ziebart, "Policy-conditioned uncertainty sets for robust Markov decision processes," in *Advances in Neural Information Processing Systems*, pages 8939–8949, 2018.

[30] Marek Petrik and Reazul Hasan Russell, "Beyond confidence regions: Tight Bayesian ambiguity sets for robust MDPs," *arXiv preprint arXiv:1902.07605*, 2019.

[31] Shiau Hong Lim, Huan Xu, and Shie Mannor, "Reinforcement learning in robust Markov decision processes," *Math. Oper. Res.*, 41(4):1325–1353, 2016.

[32] Aurko Roy, Huan Xu, and Sebastian Pokutta, "Reinforcement learning under model mismatch," in *Advances in Neural Information Processing Systems*, pages 3043–3052, Long Beach, CA, USA, 4–9 December 2017.

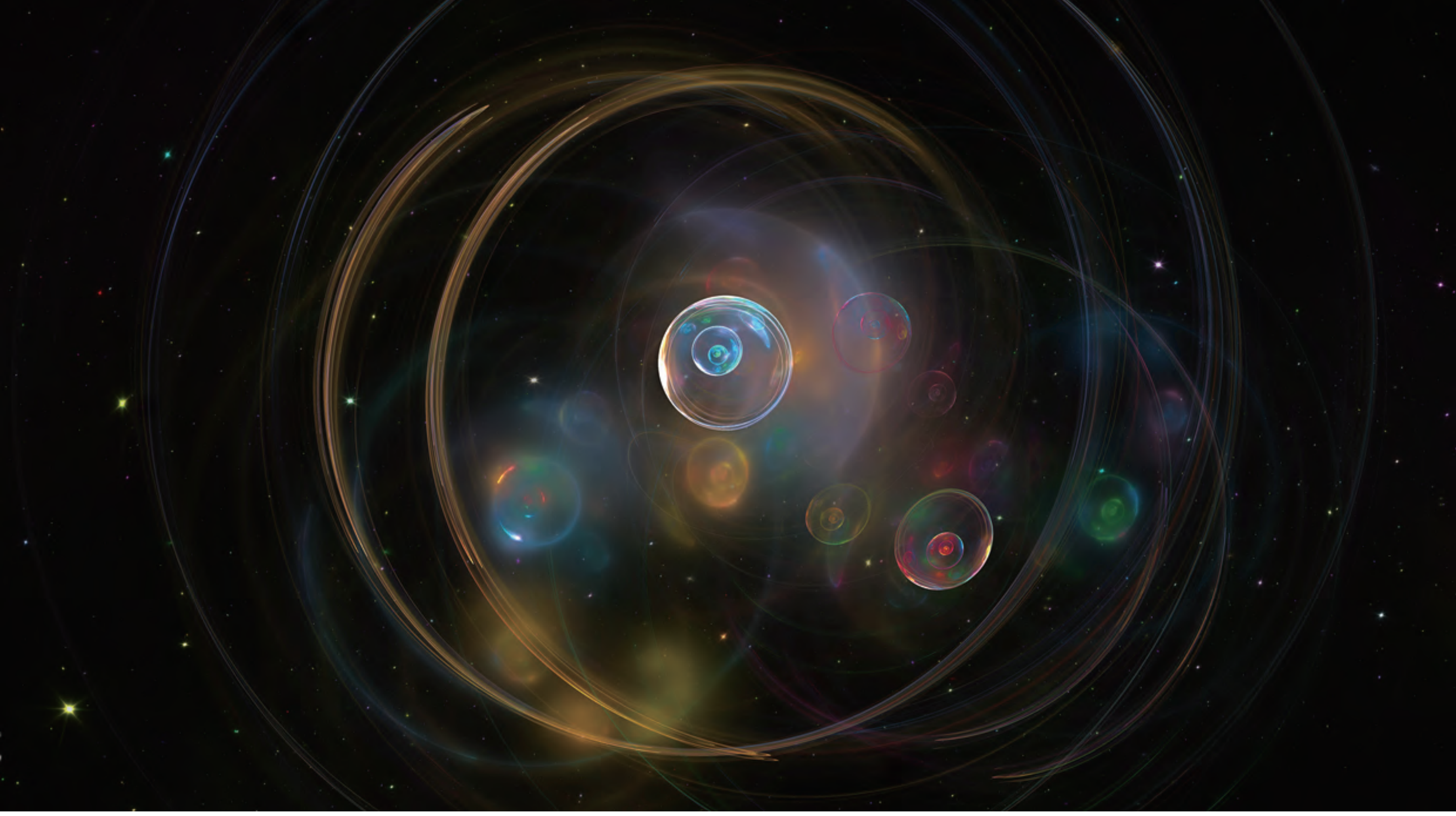
[33] Esther Derman, Daniel J. Mankowitz, Timothy A. Mann, and Shie Mannor, "A Bayesian approach to robust reinforcement learning," in *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence*, page 228, Tel Aviv, Israel, July 22–25 2019.

[34] Shiau Hong Lim and Arnaud Autef, "Kernel-based reinforcement learning in robust Markov decision processes," in *Proceedings of International Conference on Machine Learning*, pages 3973–3981, Long Beach, CA, USA, 9–15 June 2019.

[35] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *International Conference on Intelligent Robots and Systems*, pages 23–30, Vancouver, BC, Canada, 24–28 Sep 2017. IEEE.

[36] Bhairav Mehta, Manfred Diaz, Florian Golemo, Christopher J. Pal, and Liam Paull, "Active domain randomization," *arXiv preprint arXiv:1904.04762*, 2019.

[37] Yevgen Chebotar, Ankur Handa, Viktor Makoviychuk, Miles Macklin, Jan Issac, Nathan D. Ratliff, and Dieter Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience," in *International Conference on Robotics and Automation*, pages 8973–8979, Montreal, QC, Canada, May 20–24 2019. IEEE.



Random Tensor Theory, Algorithms and Applications

Mohamed El Amine Seddik, Maxime Guillaud

Abstract

Tensors have received a lot of attention recently due to their various applications and their generality property. Yet, our understanding of their theoretical properties is still limited and many of the ideas developed for matrices do not easily extend to high-dimensional tensors. In this paper, we present some basic tensor properties along with a set of classical tensor decomposition methods. We further provide a brief survey of the applications of such methods in the field of machine learning. In this context, numerous applications of tensor methods rely on extracting low-rank tensor structures from raw high-dimensional data. This motivates the analysis of random tensors with a hidden low-rank structure (also known as *spiked random tensor models*). We also present some recent theoretical advances describing the behavior of such models. Finally, we discuss the application of random tensor theory to the performance characterization of supervised and unsupervised learning with tensors.

1 Introduction

In the fields of signal processing and machine learning, there are many techniques that rely on retrieving latent (e.g. low-rank) structures from raw data. The idea of exploiting the low-rank structure of matrices started in 1904 with the work of the psychologist Charles Spearman [1] on his theory of intelligence. He suggested that there are two types of intelligence: *eductive* which defines the ability to make sense out of complexity, and *reproductive*, which represents the ability to store and reproduce information. To test his theory, he invented the notion of *factor analysis*, which consists of decomposing a given matrix into a sum of rank one matrices. Specifically, Spearman considered a 10×1000 matrix where each column (associated to a student) corresponds to the student's grades in 10 tests. He found that such a matrix decomposes into a sum of *two* rank-one components, which he interpreted as the two types of intelligence.

Matrix decomposition suffers from the lack of uniqueness — a fundamental limitation. Indeed, given a rank-*r* matrix $\boldsymbol{M} = \sum_{i=1}^{r} \boldsymbol{a}_i \otimes \boldsymbol{b}_i = \boldsymbol{A}\boldsymbol{B}^\top$, where the $\{\boldsymbol{a}_i\}_{i\in[r]}$ and $\{\boldsymbol{b}_i\}_{i\in[r]}$ are the column vectors of $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively, and $\otimes$ denotes the outer product, the decomposition of $\boldsymbol{M}$ is not unique unless other constraints are imposed on the components. This can be demonstrated by noticing that

$$\boldsymbol{M} = \boldsymbol{A}\boldsymbol{B}^\top = (\boldsymbol{A}\boldsymbol{R})(\boldsymbol{R}^{-1}\boldsymbol{B}^\top)$$

for any invertible matrix $\boldsymbol{R}$. Interestingly, multi-way arrays (*tensors*) that generalize matrices (which have two modes: columns and rows) to more than two modes yield more favorable properties, in the sense that the unique decomposition property holds under milder conditions [2]. For instance, for tensors of sub-generic rank (i.e., those that contain data arising from a structured model), the low-rank tensor decomposition is essentially unique. This is in contrast to the matrix case. In such a case, for instance, the QR or the singular value decompositions, the orthogonality conditions imposed on the factors in order to make matrix decompositions unique are merely technical and do not always correspond to a meaningful constraint in the considered problem. No such orthogonality conditions are required to make the tensor decomposition unique.

The uniqueness properties of low-rank tensor decomposition can be leveraged in the context of blind source separation, where they can work with relaxed assumptions compared to the classical approaches (in particular, much shorter data samples than methods based on high-order statistics [3]), and with a larger number of sources. The blind source separation capability is also at the core of the non-coherent, non-orthogonal multi-user wireless communication approach proposed in [4].

Tensor methods also find applications in the broader domain of machine learning. This will be the main focus of this paper as detailed in Section 3. But before we explore that, we will first recall some classical tensor operations and decomposition algorithms in Section 2. Then in Section 4, we will provide some recent findings about the analysis of *random* tensors. Finally, in Section 5, we will present a direct application of these findings to supervised and unsupervised learning where we highlight and quantify the benefit of using tensor methods on a simple framework.

2 Tensors and Decomposition Algorithms

In its simplest incarnation, a tensor can be construed as a *D*-dimensional (order) data structure. This is, akin to the generalization of vectors (1 dimension) and matrices (2 dimensions) to more than 2 dimensions (In classical linear algebra, vectors are single-dimensional data structures, whereas matrices have two dimensions: rows and columns.) The resulting *D*-way array can be interpreted either as the representation of a multilinear application (as in Newtonian physics, where coordinate-free tensor representations are used to model physical laws), or as a data structure that is naturally indexed by *D* dimensions (or modes) [5]. The tensor decomposition problem was first introduced in the late 1920s [6]. Interestingly, it is deeply rooted in experimental sciences: the decomposition of a given tensor into the sum of rank-1 components (known as the canonical polyadic, or CP, decomposition) has practical significance in numerous applications because it reveals the internal structure of the data. Indeed, low-rank tensor decomposition was re-discovered in the field of psychometrics in the 1970s where it was given the name PARAFAC [7], and then again in the field of chemometrics in the 1980s [8]. Tensor algebra is underpinned by a profound mathematical theory. Despite the relatively familiar setup provided by the analogy with the matrix case, the properties of the CP decomposition depart in major ways from the intuitions available in the matrix case [9]. For example, high-order ($D \geq 3$) tensors can have high rank even for moderate tensor sizes[1]. Unlike for matrices where the rank is bounded by the minimum of the row and column dimensions, the expected rank of a generic tensor can be greater than the dimension along each mode — in fact, the expected rank scales super-linearly [2].

Contemporary extensions of tensor theory tackle increasingly complex setups, such as the case of coupled CP decompositions [10] (where the coupling arises from the considered application), or the storage-efficient tensor-train decomposition [11] applicable to large size, high-order tensor problems. Recent theoretical developments include the study of random tensors, focusing in particular on spiked models. In the context of such models, the goal is to analytically characterize the conditions under which a low-rank informative component can be reliably separated from additive measurement noise through low-rank tensor approximation, and to predict the achievable accuracy. One approach is to make use of tools from statistical physics [12]. Attempts at developing a clean-slate spectral theory (singular vectors) of random tensors — a notoriously difficult problem — are also emerging [13]. The generalization capability of low-rank tensor decomposition is also being investigated in the context of "missing data" formulations [14]. In the remainder of this section, we will present some basic notions of tensor algebra, as well as classical decomposition algorithms.

2.1 Tensor Operations

Here, we recall some tensor notations and operations that will be used throughout the paper.

Symbolic notations:

- The set $\{1, \ldots, n\}$ is denoted as $[n]$
- Scalars are denoted by lowercase letters as *a*, *b*, *c*
- Vectors are denoted by bold lowercase letters as ***a***, ***b***, ***c***
- Matrices are denoted by bold uppercase letters as ***A***, ***B***, ***C***
- Tensors are denoted as **A**, **B**, **C**
- The entry $(i_1, \ldots, i_d)$ of the tensor **T** is denoted as $T_{i_1,\ldots,i_d}$
- The scalar product between $\boldsymbol{u}$ and $\boldsymbol{v}$ is denoted as $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_i u_i v_i$
- The ℓ_2-norm of a vector $\boldsymbol{u}$ is denoted as $\|\boldsymbol{u}\|^2 = \langle \boldsymbol{u}, \boldsymbol{u} \rangle$

[1] For instance, a tensor of dimensions 5×5×5×5 can have rank 37, while a matrix of dimensions 25×25 which contains the same number of scalar coefficients as the tensor, has rank 25 at most.

- The normal distribution with mean m and variance σ^2 is denoted as $\mathcal{N}(m, \sigma^2)$
- $Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{t^2}{2}} dt$ corresponds to the Gaussian tail function
- $\xrightarrow{\text{a.s.}}$ stands for the almost sure convergence and $\xrightarrow{\mathcal{D}}$ for the convergence in distribution
- The d-dimensional unit sphere is denoted as $\mathbb{S}^{d-1}$

Inner product and norm: The inner product of two same-sized order D tensors $\mathbf{T}, \mathbf{T}' \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ is the sum of the products of their entries and is denoted as

$$\langle \mathbf{T}, \mathbf{T}' \rangle = \sum_{i_1,\ldots,i_D} T_{i_1 \cdots i_D} T'_{i_1 \cdots i_D} \quad (1)$$

The norm $\|\mathbf{T}\|$ of $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ is given as $\|\mathbf{T}\|^2 = \langle \mathbf{T}, \mathbf{T} \rangle$.

Rank-one tensors: A rank-1 tensor of order D is defined in a way analogous to the matrix case, as the outer product of D vectors of appropriate dimensions (while a rank-1 matrix is defined by the outer product of only two vectors: a column vector and a row vector). For D = 3, a rank-1 tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ is the outer product of 3 vectors $\boldsymbol{a} \in \mathbb{R}^{d_1}$, $\boldsymbol{b} \in \mathbb{R}^{d_2}$ and $\boldsymbol{c} \in \mathbb{R}^{d_3}$. As such, we write

$$\mathbf{T} = \boldsymbol{a} \otimes \boldsymbol{b} \otimes \boldsymbol{c} \quad (2)$$

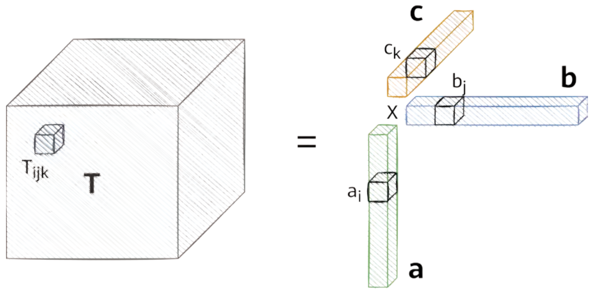


Figure 1 Rank-1 third-order tensor. $\mathbf{T} = \boldsymbol{a} \otimes \boldsymbol{b} \otimes \boldsymbol{c}$.
The (i, j, k)-th entry of **T** is given by $T_{ijk} = a_i b_j c_k$.

or entry-wise $T_{ijk} = a_i b_j c_k$. Figure 1 provides an illustration for a rank-1 tensor of order $D = 3$.

More generally, an order D tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ is said to be a *rank-one* tensor if it can be written as the outer product of D vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_D$ of respective dimensions $d_1, \ldots, d_D$. Specifically,

$$\mathbf{T} = \bigotimes_{j=1}^{D} \boldsymbol{a}_j = \boldsymbol{a}_1 \otimes \cdots \otimes \boldsymbol{a}_D \quad (3)$$

where the outer product $\bigotimes$ is defined such that $\left(\bigotimes_{j=1}^D \boldsymbol{a}_j\right)_{i_1 \ldots i_D} = \prod_{j=1}^D (\boldsymbol{a}_j)_{i_j}$, i.e., each element of the rank-one tensor is the product of the corresponding vectors entries.

Tensor multiplication: The j-mode tensor product between two tensors $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ and $\mathbf{T}' \in \mathbb{R}^{d'_1 \times \cdots \times d'_{D'}}$ (with $d_j = d'_j$) is denoted by $\mathbf{T} \times_j \mathbf{T}'$ which is a tensor of size $d_1 \times \cdots \times d_{j-1} \times d_{j+1} \times \cdots \times d_D \times d'_1 \times \cdots \times d'_{j-1} \times d'_{j+1} \times \cdots \times d'_{D'}$ whose entries are given by

$$(\mathbf{T} \times_j \mathbf{T}')_{i_1 \cdots i_{j-1} i_{j+1} \cdots i_D i'_1 \cdots i'_{j-1} i'_{j+1} \cdots i'_{D'}} = \sum_{i_j=1}^{d_j} T_{i_1 \cdots i_D} T'_{i'_1 \cdots i'_{D'}}$$

In the case where $\mathbf{T}'$ is a matrix that we denote as $\boldsymbol{M} \in \mathbb{R}^{m \times d_j}$, the (matrix) product of a tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ with $\boldsymbol{M}$ is denoted by $\mathbf{T} \times_j \boldsymbol{M}$ and is a tensor of size $d_1 \times \cdots \times d_{j-1} \times m \times d_{j+1} \times \cdots \times d_D$. Entry-wise, the j-mode (matrix) product is defined as

$$(\mathbf{T} \times_j \boldsymbol{M})_{i_1 \cdots i_{j-1} k i_{j+1} \cdots i_D} = \sum_{i_j=1}^{d_j} T_{i_1 \cdots i_D} M_{k i_j} \tag{4}$$

Similarly, the j-mode (vector) product or *contraction* of an order D tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ with a vector $\boldsymbol{v} \in \mathbb{R}^{d_j}$ is denoted by $\mathbf{T} \times_j \boldsymbol{v}$ and results in a tensor of order $D-1$ of dimension $d_1 \times \cdots \times d_{j-1} \times d_{j+1} \times \cdots \times d_D$. Element-wise, the j-mode contraction is defined as

$$(\mathbf{T} \times_j \boldsymbol{v})_{i_1 \cdots i_{j-1} i_{j+1} \cdots i_D} = \sum_{i_j=1}^{d_j} T_{i_1 \cdots i_D} v_{i_j} \tag{5}$$

which, essentially, involves computing the inner product of each mode-j *fiber* — seeing the tensor as stacked vectors (fibers) along its j-th mode — with the vector $\boldsymbol{v}$.

Tensor unfolding: The j-mode *unfolding* or matricization of a tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ is denoted by $\mathrm{Mat}_j(\mathbf{T})$ and involves arranging the mode-j slices of $\mathbf{T}$ to be the block columns of the resulting matrix. Entry-wise, $\mathrm{Mat}_j(\mathbf{T})$ is constructed as

$$[\mathrm{Mat}_j(\mathbf{T})]_{i_j, k} = T_{i_1, \ldots, i_D} \tag{6}$$

where $k = 1 + \sum_{\ell \neq j}^{D} (i_\ell - 1) J_\ell$ with $J_\ell = \prod_{m \neq j}^{\ell - 1} d_m$. Figure 2 depicts the unfolding operation of a third-order tensor.

Tensor Rank and the CANDECOMP/PARAFAC Decomposition (CPD): Given a third-order tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$, the CP decomposition [6, 15] involves decomposing $\mathbf{T}$ into a sum of rank-one tensors with the minimal number of terms. Specifically,

$$\mathbf{T} = \sum_{i=1}^{R} \boldsymbol{a}_i \otimes \boldsymbol{b}_i \otimes \boldsymbol{c}_i \tag{7}$$

The rank of $\mathbf{T}$ denoted by $\mathrm{rank}(\mathbf{T})$ is defined as the smallest integer R for which $\mathbf{T}$ decomposes as above. Figure 3 depicts the CP decomposition of a third-order tensor. More generally, a CP decomposition of a D-order tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ is written as

$$\mathbf{T} = \sum_{i=1}^{R} \bigotimes_{j=1}^{D} \boldsymbol{a}_j^{(i)} = \sum_{i=1}^{R} \boldsymbol{a}_1^{(i)} \otimes \cdots \otimes \boldsymbol{a}_D^{(i)} \tag{8}$$

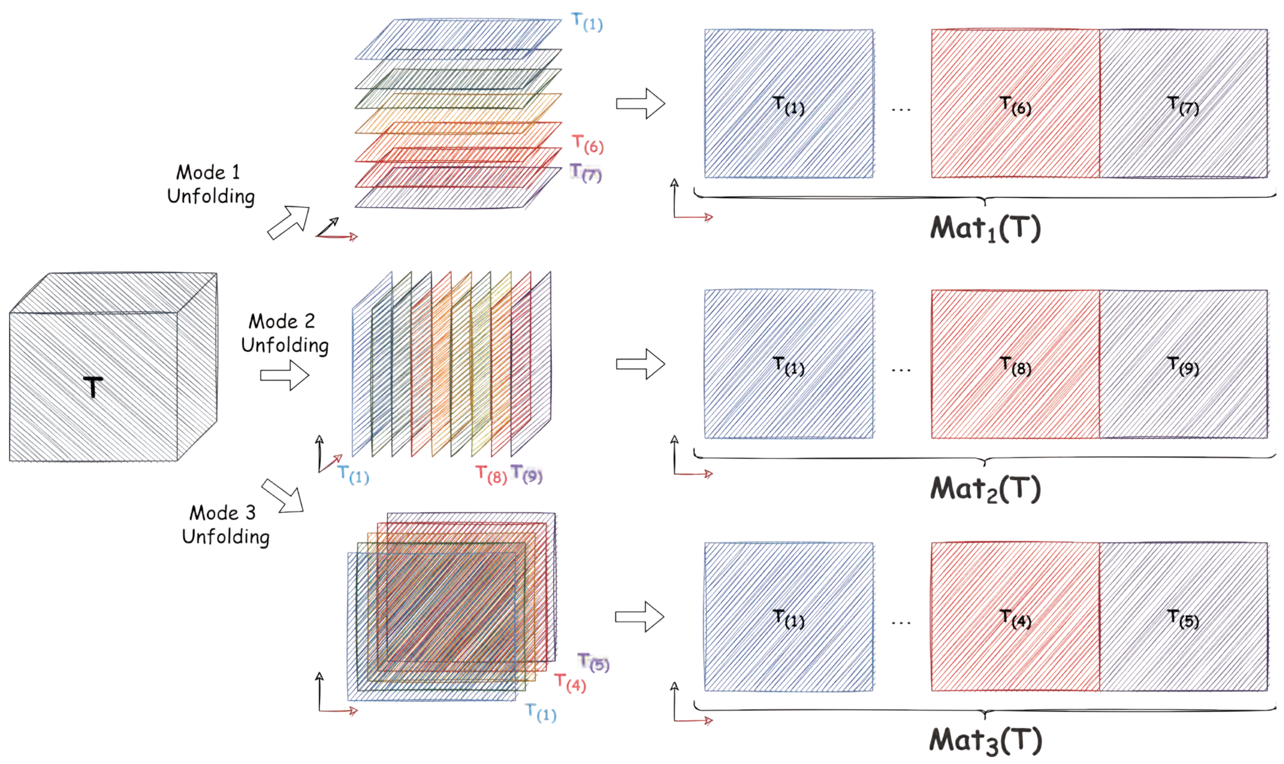


Figure 2 Unfolding of a third-order tensor along its three modes

Tucker decomposition: The Tucker decomposition [16] is a generalization of the standard principal component analysis (PCA) to higher-order arrays. Essentially, it involves decomposing a given tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ into a *core* tensor $\mathbf{G} \in \mathbb{R}^{R_1 \times \cdots \times R_D}$ multiplied mode-wise by a matrix of size $d_i \times R_i$ for each $i \in [D]$. For a third-order tensor (illustrated in Figure 4), we have

$$\begin{aligned} \mathbf{T} &= \mathbf{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C} \\ &= \sum_{i_1=1}^{R_1} \sum_{i_2=1}^{R_2} \sum_{i_3=1}^{R_3} G_{i_1 i_2 i_3} \boldsymbol{a}_{i_1} \otimes \boldsymbol{b}_{i_2} \otimes \boldsymbol{c}_{i_3} \end{aligned} \tag{9}$$

where $\boldsymbol{A} \in \mathbb{R}^{d_1 \times R_1}, \boldsymbol{B} \in \mathbb{R}^{d_2 \times R_2}$ and $\boldsymbol{C} \in \mathbb{R}^{d_3 \times R_3}$ with $\boldsymbol{a}_i, \boldsymbol{b}_j$ and $\boldsymbol{c}_k$ are the column vectors of $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{C}$ respectively. Entry-wise, we have

$$T_{ijk} = \sum_{i_1=1}^{R_1} \sum_{i_2=1}^{R_2} \sum_{i_3=1}^{R_3} G_{i_1 i_2 i_3} A_{i i_1} B_{j i_2} C_{k i_3} \tag{10}$$

Tensor train decomposition: Tensor networks are a more general type of tensor decompositions. The most classical of these is known as tensor train (TT) decomposition, which consists in *compressing* a given tensor $\mathbf{T}$ in terms of a set of smaller tensors. The TT decomposition takes the following form

$$T_{i_1,\ldots,i_D} = \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \cdots \sum_{r_{D-1}=1}^{R_{D-1}} G^{(1)}_{i_1 r_1} G^{(2)}_{r_1 i_2 r_2} \cdots G^{(D)}_{r_{D-1}, i_D}$$

where the $\mathbf{G}^{(i)}$ for $i \in [D]$ are the factors to optimize. Figure 5 illustrates the TT decomposition of an order D tensor $\mathbf{T}$.

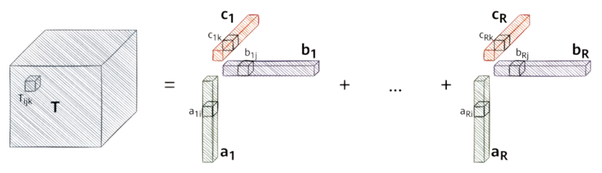


Figure 3 CP decomposition of a rank-*R* third-order tensor

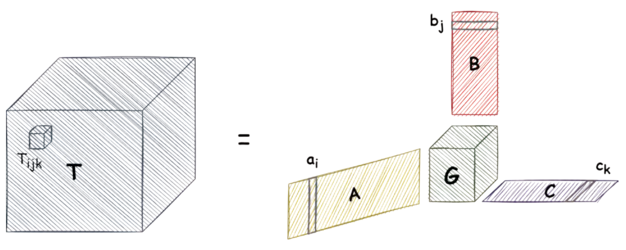


Figure 4 Tucker decomposition of a third-order tensor

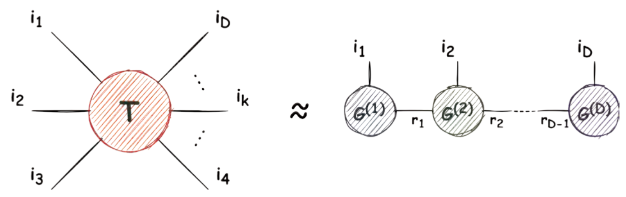


Figure 5 TT decomposition of a *D*-order tensor

2.2 Tensor Decomposition Algorithms

The aim of CP decomposition algorithms is to identify the R rank-one components in equation 8 given $\mathbf{T}$. However, even in the presence of low-rank data, observation noise typically increases the rank of the observation beyond R. In that case, if R is known a priori, one can attempt to find a rank-R approximation of a given generic tensor. Specifically, for a given third-order tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$, we would like to approximate it by $\mathbf{M}$ such that

$$\mathbf{T} \approx \mathbf{M} = \sum_{i=1}^{R} \boldsymbol{a}_i \otimes \boldsymbol{b}_i \otimes \boldsymbol{c}_i \tag{11}$$

where $\boldsymbol{a}_i \in \mathbb{R}^{d_1}, \boldsymbol{b}_i \in \mathbb{R}^{d_2}$ and $\boldsymbol{c}_i \in \mathbb{R}^{d_3}$. The goal is to identify the best approximation[2], often in the sense of minimizing the mean squared error between $\mathbf{T}$ and $\mathbf{M}$. The integer R is fixed by some initial procedure, and coincides with the rank of $\mathbf{T}$ if the approximation is consistent. We will present in the sequel various tensor decomposition algorithms that provide such approximation.

2.2.1 Rank-One Approximation Methods

Let us first focus on the rank-one case. Suppose that we are given a rank-one tensor $\mathbf{T} = \boldsymbol{a} \otimes \boldsymbol{b} \otimes \boldsymbol{c} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and we want to recover its components $\boldsymbol{a} \in \mathbb{R}^{d_1}, \boldsymbol{b} \in \mathbb{R}^{d_2}$ and $\boldsymbol{c} \in \mathbb{R}^{d_3}$. These components are assumed to be of unit Euclidean norm for simplicity.

Tensor unfolding: The most basic approach involves unfolding the tensor $\mathbf{T}$ into a rectangular matrix and then computing its components via singular value decomposition (SVD). Indeed, unfolding $\mathbf{T}$ along each of the D dimensions yields the following linearizations:

[2] Note that the best approximation does not always exist. This is because the Eckart-Young theorem does not generalize from matrices to tensors in a straightforward manner. For details, see [17, 18].

$$\mathrm{Mat}_1(\mathbf{T}) = \boldsymbol{a}\,\mathrm{vec}(\boldsymbol{b}\otimes\boldsymbol{c})^\top, \quad \mathrm{Mat}_2(\mathbf{T}) = \boldsymbol{b}\,\mathrm{vec}(\boldsymbol{a}\otimes\boldsymbol{c})^\top,$$
$$\mathrm{Mat}_3(\mathbf{T}) = \boldsymbol{c}\,\mathrm{vec}(\boldsymbol{a}\otimes\boldsymbol{b})^\top.$$

Therefore, the components $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$ can be recovered as the dominant left singular vector of $\mathrm{Mat}_1(\mathbf{T})$, $\mathrm{Mat}_2(\mathbf{T})$ and $\mathrm{Mat}_3(\mathbf{T})$ respectively. The tensor unfolding method is described in Algorithm 1.

Algorithm 1 Tensor unfolding [19]

```
Require: An order 3 tensor $\mathbf{T} \in \mathbb{R}^{d_1\times d_2\times d_3}$.
Output: Rank-one approximation of $\mathbf{T}$.
  for $i \in [3]$ do
    Set $\boldsymbol{v}_i$ the dominant left singular vector of $\mathrm{Mat}_i(\mathbf{T})$
  end for
  $\hat{\boldsymbol{a}} \leftarrow \boldsymbol{v}_1, \quad \hat{\boldsymbol{b}} \leftarrow \boldsymbol{v}_2, \quad \hat{\boldsymbol{c}} \leftarrow \boldsymbol{v}_3$
```

Tensor power iteration: Finding the best rank-one approximation of $\mathbf{T}$ can be formulated via a variational approach [20] that involves optimizing the following objective:

$$\underset{\|\boldsymbol{u}\|=1,\|\boldsymbol{v}\|=1,\|\boldsymbol{w}\|=1}{\arg\max} \quad |\langle \mathbf{T}, \boldsymbol{u}\otimes\boldsymbol{v}\otimes\boldsymbol{w}\rangle| \tag{12}$$

Writing the KKT conditions of the above optimization problem allows us to generalize the concept of singular values and vectors to tensors. Specifically, the singular vectors $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ and the singular value $\lambda > 0$ of $\mathbf{T}$ are defined as the solutions of the system of equations

$$\begin{gathered}\mathbf{T}\times_2 \boldsymbol{v}\times_3 \boldsymbol{w} = \lambda\boldsymbol{u}, \quad \mathbf{T}\times_1 \boldsymbol{u}\times_3 \boldsymbol{w} = \lambda\boldsymbol{v},\\ \mathbf{T}\times_1 \boldsymbol{u}\times_2 \boldsymbol{v} = \lambda\boldsymbol{w}.\end{gathered} \tag{13}$$

In practice, as for matrices, the components ($\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$) of $\mathbf{T}$ can be estimated via power iteration by alternating the following operations (starting from a random initialization):

$$\hat{\boldsymbol{a}} \leftarrow \frac{\mathbf{T}\times_2 \hat{\boldsymbol{b}}\times_3 \hat{\boldsymbol{c}}}{\|\mathbf{T}\times_2 \hat{\boldsymbol{b}}\times_3 \hat{\boldsymbol{c}}\|}, \quad \hat{\boldsymbol{b}} \leftarrow \frac{\mathbf{T}\times_1 \hat{\boldsymbol{a}}\times_3 \hat{\boldsymbol{c}}}{\|\mathbf{T}\times_1 \hat{\boldsymbol{a}}\times_3 \hat{\boldsymbol{c}}\|},$$
$$\hat{\boldsymbol{c}} \leftarrow \frac{\mathbf{T}\times_1 \hat{\boldsymbol{a}}\times_2 \hat{\boldsymbol{b}}}{\|\mathbf{T}\times_1 \hat{\boldsymbol{a}}\times_2 \hat{\boldsymbol{b}}\|}$$

The tensor power iteration method is described in Algorithm 2. We will discuss the convergence properties and accuracy of this method in Section 4.

Algorithm 2 Tensor power iteration [21]

```
Require: An order 3 tensor $\mathbf{T} \in \mathbb{R}^{d_1\times d_2\times d_3}$ and initialization components $\boldsymbol{a}_0, \boldsymbol{b}_0, \boldsymbol{c}_0$.
Output: Rank-one approximation of $\mathbf{T}$.
  $(\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, \hat{\boldsymbol{c}}) \leftarrow (\boldsymbol{a}_0, \boldsymbol{b}_0, \boldsymbol{c}_0)$
  while Not converged do
    $\hat{\boldsymbol{a}} \leftarrow \frac{\mathbf{T}\times_2\hat{\boldsymbol{b}}\times_3\hat{\boldsymbol{c}}}{\|\mathbf{T}\times_2\hat{\boldsymbol{b}}\times_3\hat{\boldsymbol{c}}\|}, \quad \hat{\boldsymbol{b}} \leftarrow \frac{\mathbf{T}\times_1\hat{\boldsymbol{a}}\times_3\hat{\boldsymbol{c}}}{\|\mathbf{T}\times_1\hat{\boldsymbol{a}}\times_3\hat{\boldsymbol{c}}\|}, \quad \hat{\boldsymbol{c}} \leftarrow \frac{\mathbf{T}\times_1\hat{\boldsymbol{a}}\times_2\hat{\boldsymbol{b}}}{\|\mathbf{T}\times_1\hat{\boldsymbol{a}}\times_2\hat{\boldsymbol{b}}\|}$
  end while
```

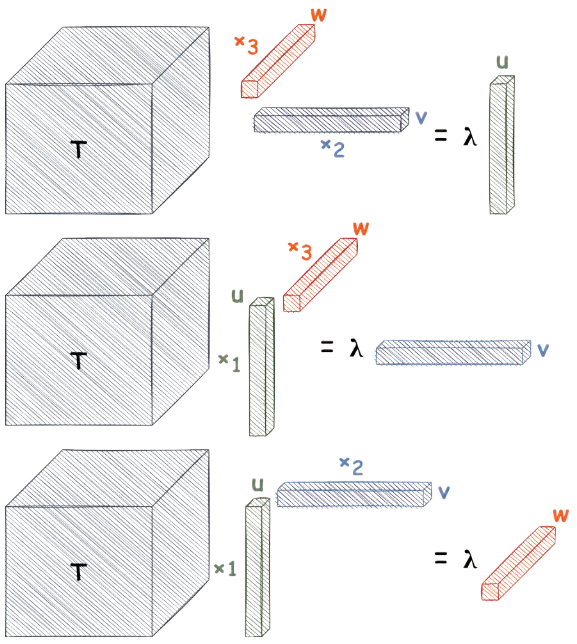


Figure 6 Singular values and vectors of a third-order tensor

Newton method: The newton method mainly involves finding the zero of a given function through recursive Newton iterations [22]. Computing the singular vectors of a tensor can be reformulated in terms of finding the zero of the following mapping

$$G : (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \mapsto \mathrm{concat}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) - F(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w})$$

where $\mathrm{concat}(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) = [\boldsymbol{u}^\top, \boldsymbol{v}^\top, \boldsymbol{w}^\top]^\top \in \mathbb{R}^{d_1+d_2+d_3}$ is the concatenation mapping and

$F : (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \mapsto \mathrm{concat}(\mathbf{T}\times_2 \boldsymbol{v}\times_3 \boldsymbol{w}, \mathbf{T}\times_1 \boldsymbol{u}\times_3 \boldsymbol{w}, \mathbf{T}\times_1 \boldsymbol{u}\times_2 \boldsymbol{v})$.

Therefore, the Jacobian of G is

$$\nabla G(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) = \begin{bmatrix} \boldsymbol{I}_{p1} & -\mathbf{T}\times_3 \boldsymbol{w} & -\mathbf{T}\times_2 \boldsymbol{v} \\ -(\mathbf{T}\times_3 \boldsymbol{w})^\top & \boldsymbol{I}_{p2} & -\mathbf{T}\times_1 \boldsymbol{u} \\ -(\mathbf{T}\times_2 \boldsymbol{v})^\top & -(\mathbf{T}\times_1 \boldsymbol{u})^\top & \boldsymbol{I}_{p3} \end{bmatrix}$$

Hence, the Newton iterations are given as

$$\begin{aligned}&\mathrm{concat}(\boldsymbol{u}^{(t+1)}, \boldsymbol{v}^{(t+1)}, \boldsymbol{w}^{(t+1)}) \leftarrow \mathrm{concat}(\boldsymbol{u}^{(t)}, \boldsymbol{v}^{(t)}, \boldsymbol{w}^{(t)})\\ &- \nabla G(\boldsymbol{u}^{(t)}, \boldsymbol{v}^{(t)}, \boldsymbol{w}^{(t)})^{-1} G(\boldsymbol{u}^{(t)}, \boldsymbol{v}^{(t)}, \boldsymbol{w}^{(t)})\end{aligned}$$

followed by a scaling of the vectors $\boldsymbol{u}^{(t+1)}, \boldsymbol{v}^{(t+1)}, \boldsymbol{w}^{(t+1)}$ to be of unit norm. The Newton method is described in Algorithm 3.

Algorithm 3 Newton method for best rank-one approximation [22]

```
Require: An order 3 tensor $\mathbf{T} \in \mathbb{R}^{d_1\times d_2\times d_3}$ and initialization components $\boldsymbol{a}_0, \boldsymbol{b}_0, \boldsymbol{c}_0$.
Output: Rank-one approximation of $\mathbf{T}$.
  $(\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, \hat{\boldsymbol{c}}) \leftarrow (\boldsymbol{a}_0, \boldsymbol{b}_0, \boldsymbol{c}_0)$
  while Not converged do
    $\mathrm{concat}(\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, \hat{\boldsymbol{c}}) \leftarrow \mathrm{concat}(\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, \hat{\boldsymbol{c}}) - \nabla G(\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, \hat{\boldsymbol{c}})^{-1} G(\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, \hat{\boldsymbol{c}})$
    $\hat{\boldsymbol{a}} \leftarrow \frac{\boldsymbol{a}}{\|\boldsymbol{a}\|}, \quad \hat{\boldsymbol{b}} \leftarrow \frac{\boldsymbol{b}}{\|\boldsymbol{b}\|}, \quad \hat{\boldsymbol{c}} \leftarrow \frac{\boldsymbol{c}}{\|\boldsymbol{c}\|}$
  end while
```

2.2.2 Low-Rank Approximation Methods

Let us now consider a more general case with a given rank-R tensor $\mathbf{T} = \sum_{i=1}^{R} \lambda_i \boldsymbol{a}_i \otimes \boldsymbol{b}_i \otimes \boldsymbol{c}_i \in \mathbb{R}^{d_1 \times d_2 \times d_3}$, for which we seek to recover the normalized components $\boldsymbol{a}_i \in \mathbb{R}^{d_1}, \boldsymbol{b}_i \in \mathbb{R}^{d_2}$ and $\boldsymbol{c}_i \in \mathbb{R}^{d_3}$ for all $i \in [R]$ from the observation $\mathbf{T}$. The scalars $\lambda_1 \geq \cdots \geq \lambda_R$ are the generalization of the concept of singular values for tensors.

Deflation algorithm: The first approach involves performing iterative rank-one approximations of the tensor $\mathbf{T}$ [23], and subtracting the identified rank-one component at each iteration. Specifically, given some algorithm (mapping) $\phi : \mathbb{R}^{d_1 \times d_2 \times d_3} \to \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \times \mathbb{R}^{d_3}$ that outputs the normalized components of the best rank-one approximation of a tensor: for the tensor $\mathbf{T}$ above, $(\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{b}}_1, \hat{\boldsymbol{c}}_1) = \phi(\mathbf{T})$ with $\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{b}}_1, \hat{\boldsymbol{c}}_1$ being estimates of $\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1$ respectively. The deflation method is detailed in Algorithm 4.

Algorithm 4 Deflation algorithm [23]

Require: An order 3 tensor T tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$, rank R and a best rank-one approximation method ϕ.

Output: Rank-R approximation of $\mathbf{T}$.

```
  $\mathbf{Y} \leftarrow \mathbf{T}$
  for $i \in [R]$ do
      $(\hat{\boldsymbol{a}}_i, \hat{\boldsymbol{b}}_i, \hat{\boldsymbol{c}}_i) = \phi(\mathbf{Y})$
      $\hat{\lambda}_i \leftarrow \langle \mathbf{Y}, \hat{\boldsymbol{a}}_i \otimes \hat{\boldsymbol{b}}_i \otimes \hat{\boldsymbol{c}}_i \rangle$
      $\mathbf{X}_i \leftarrow \hat{\lambda}_i \hat{\boldsymbol{a}}_i \otimes \hat{\boldsymbol{b}}_i \otimes \hat{\boldsymbol{c}}_i$
      $\mathbf{Y} \leftarrow \mathbf{Y} - \mathbf{X}_i$
  end for
  $\mathbf{E} \leftarrow \mathbf{Y}$
  while Not converged do
  for $i \in [R]$ do
      $\mathbf{E} \leftarrow \mathbf{E} + \mathbf{X}_i$
      $(\hat{\boldsymbol{a}}_i, \hat{\boldsymbol{b}}_i, \hat{\boldsymbol{c}}_i) = \phi(\mathbf{E})$
      $\hat{\lambda}_i \leftarrow \langle \mathbf{Y}, \hat{\boldsymbol{a}}_i \otimes \hat{\boldsymbol{b}}_i \otimes \hat{\boldsymbol{c}}_i \rangle$
      $\mathbf{X}_i \leftarrow \hat{\lambda}_i \hat{\boldsymbol{a}}_i \otimes \hat{\boldsymbol{b}}_i \otimes \hat{\boldsymbol{c}}_i$
      $\mathbf{E} \leftarrow \mathbf{E} - \mathbf{X}_i$
  end for
end while
```

Alternating least squares (ALS): This method [9] attempts to solve equation 11 by minimizing the mean squared error between $\mathbf{T}$ and $\mathbf{M}$ — considered a non-convex optimization problem — as follows

$$\min_{\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}} \sum_{ijk} (T_{ijk} - M_{ijk})^2 = \min_{\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}} \sum_{ijk} \left(T_{ijk} - \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr} \right)^2$$

where $\boldsymbol{M}$ is constructed in terms of the three matrices $\boldsymbol{A} \in \mathbb{R}^{d_1 \times R}, \boldsymbol{B} \in \mathbb{R}^{d_2 \times R}$ and $\boldsymbol{C} \in \mathbb{R}^{d_3 \times R}$ with columns $\boldsymbol{a}_i$, $\boldsymbol{b}_i$ and $\boldsymbol{c}_i$ respectively. The ALS method involves optimizing these matrices in an alternative manner by rewriting the objective for each matrix as shown below. This allows to transform the non-convex optimization problem into many easier ones which enjoy the property of being convex (linear) problems. Indeed, the unfolding operation allows the following linearizations

$$\begin{aligned}
\min_{A} \sum_{ijk} \left(T_{ijk} - \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr} \right)^2 &= \min_{A} \| \operatorname{Mat}_1(\mathbf{T}) - \boldsymbol{A}(\boldsymbol{B} \odot \boldsymbol{C})^\top \|_{\mathrm{F}}^2 \\
\min_{B} \sum_{ijk} \left(T_{ijk} - \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr} \right)^2 &= \min_{B} \| \operatorname{Mat}_2(\mathbf{T}) - \boldsymbol{B}(\boldsymbol{A} \odot \boldsymbol{C})^\top \|_{\mathrm{F}}^2 \\
\min_{C} \sum_{ijk} \left(T_{ijk} - \sum_{r=1}^{R} a_{ir} b_{jr} c_{kr} \right)^2 &= \min_{C} \| \operatorname{Mat}_3(\mathbf{T}) - \boldsymbol{C}(\boldsymbol{A} \odot \boldsymbol{B})^\top \|_{\mathrm{F}}^2
\end{aligned}$$

where $\boldsymbol{B} \odot \boldsymbol{C}$ denotes the *Khatri-Rao* product between $\boldsymbol{B}$ and $\boldsymbol{C}$, yielding a matrix of size $(d_2 d_3) \times R$ defined as

$$\boldsymbol{B} \odot \boldsymbol{C} = \begin{bmatrix} \boldsymbol{b}_1 \circ \boldsymbol{c}_1 & \boldsymbol{b}_2 \circ \boldsymbol{c}_2 & \cdots & \boldsymbol{b}_R \circ \boldsymbol{c}_R \end{bmatrix} \tag{14}$$

and $\boldsymbol{u} \circ \boldsymbol{v}$ stands for the Kronecker product between $\boldsymbol{u}$ and $\boldsymbol{v}$. The objectives can therefore be optimized by alternatively solving the above linear systems of equations in the least-squares sense. These objectives are of the form $\arg\min_{\boldsymbol{W}} \|\boldsymbol{Y} - \boldsymbol{W}\boldsymbol{X}\|_F^2$, the solution of which in closed form is

$$\boldsymbol{W}[\boldsymbol{Y}, \boldsymbol{X}] \equiv \boldsymbol{Y}\boldsymbol{X}^\top \left(\boldsymbol{X}\boldsymbol{X}^\top \right)^{-1} \tag{15}$$

provided that $\boldsymbol{X}\boldsymbol{X}^\top$ is invertible. The ALS method is described in Algorithm 5.

Algorithm 5 Alternating Least Squares (ALS) [9]

Require: An order 3 tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$, rank R and number of iterations N_{iter}.

Output: Rank-R approximation of $\mathbf{T}$.

```
Initialize B and C randomly
for $t \in [N_{iter}]$ do
    $\boldsymbol{A} \leftarrow \boldsymbol{W}\left[\operatorname{Mat}_1(\mathbf{T}), (\boldsymbol{B} \odot \boldsymbol{C})^\top\right]$
    $\boldsymbol{B} \leftarrow \boldsymbol{W}\left[\operatorname{Mat}_2(\mathbf{T}), (\boldsymbol{A} \odot \boldsymbol{C})^\top\right]$
    $\boldsymbol{C} \leftarrow \boldsymbol{W}\left[\operatorname{Mat}_3(\mathbf{T}), (\boldsymbol{A} \odot \boldsymbol{B})^\top\right]$
end for
```

Simultaneous diagonalization algorithm (SDA): SDA is an alternative to ALS first introduced in [24] and is based on the eigendecomposition of matrices constructed through random contractions of the initial tensor. Specifically, given some random (unitary) Gaussian vector $\boldsymbol{w}^{(1)} \in \mathbb{R}^{d_3}$, the contraction of $\mathbf{T}$ with $\boldsymbol{w}^{(1)}$ is

$$\mathbf{T} \times_3 \boldsymbol{w}^{(1)} = \sum_{i=1}^{R} \langle \boldsymbol{c}_i, \boldsymbol{w}^{(1)} \rangle \boldsymbol{a}_i \otimes \boldsymbol{b}_i = \boldsymbol{A}\boldsymbol{D}_1\boldsymbol{B}^\top \tag{16}$$

where $\boldsymbol{D}_1 \in \mathbb{R}^{R\times R}$ is a diagonal matrix with diagonal entries $\langle \boldsymbol{c}_i, \boldsymbol{w}^{(1)}\rangle$ for $i \in [R]$. Sampling another random (unit-norm) Gaussian vector $\boldsymbol{w}^{(2)}$ and contracting it with $\mathbf{T}$ yields $\mathbf{T} \times_3 \boldsymbol{w}^{(2)} = \boldsymbol{A}\boldsymbol{D}_2\boldsymbol{B}^\top$ where similarly $\boldsymbol{D}_2 \in \mathbb{R}^{R\times R}$ is a diagonal matrix with diagonal entries $\langle \boldsymbol{c}_i, \boldsymbol{w}^{(2)}\rangle$ for $i \in [R]$. Assuming $\mathbf{T} \times_3 \boldsymbol{w}^{(2)}$ is invertible (otherwise consider the pseudo-inverse or the Moore-Penrose inverse denoted $\boldsymbol{M}^\dagger$), we have

$$\begin{aligned}\mathbf{T} \times_3 \boldsymbol{w}^{(1)} \left(\mathbf{T} \times_3 \boldsymbol{w}^{(2)}\right)^{-1} &= \boldsymbol{A}\boldsymbol{D}_1\boldsymbol{B}^\top \left(\boldsymbol{B}^\top\right)^{-1} \boldsymbol{D}_2^{-1}\boldsymbol{A}^{-1} \\ &= \boldsymbol{A}\boldsymbol{D}_1\boldsymbol{D}_2^{-1}\boldsymbol{A}^{-1}\end{aligned}$$

and therefore the components $\boldsymbol{a}_i$ can be recovered through eigendecomposition of $\mathbf{T} \times_3 \boldsymbol{w}^{(1)} \left(\mathbf{T} \times_3 \boldsymbol{w}^{(2)}\right)^{-1}$. The SDA procedure is detailed in Algorithm 6.

Algorithm 6 Simultaneous diagonalization algorithm [24]

```
Require: An order 3 tensor $\mathbf{T} \in \mathbb{R}^{d_1\times d_2\times d_3}$ and rank $R$.
Output: Rank-$R$ approximation of $\mathbf{T}$.
    Sample $\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)} \sim \mathcal{N}(0, d_3^{-1}\boldsymbol{I}_{p_3})$ independently.
    Set $\boldsymbol{M}_1 \leftarrow \mathbf{T} \times_3 \boldsymbol{w}^{(1)}$,    $\boldsymbol{M}_2 \leftarrow \mathbf{T} \times_3 \boldsymbol{w}^{(2)}$.
    For $i \in [R]$ set $\hat{\boldsymbol{a}}_i$ and $\hat{\boldsymbol{b}}_i$ to be the dominant eigenvectors of $\boldsymbol{M}_1(\boldsymbol{M}_2)^\dagger$ and $((\boldsymbol{M}_2)^\dagger\boldsymbol{M}_1)^\top$ respectively.
    Pair up $\hat{\boldsymbol{a}}_i, \hat{\boldsymbol{b}}_i$ according to their eigenvalues.
    Solve the linear system $\mathbf{T} = \sum_{i=1}^R \hat{\boldsymbol{a}}_i \otimes \hat{\boldsymbol{b}}_i \otimes \hat{\boldsymbol{c}}_i$ to recover the vectors $\hat{\boldsymbol{c}}_i$.
    Return the factor matrices $\boldsymbol{A} \in \mathbb{R}^{d_1\times R}, \boldsymbol{B} \in \mathbb{R}^{d_2\times R}$ and $\boldsymbol{C} \in \mathbb{R}^{d_3\times R}$.
```

Higher order singular value decomposition (HOSVD): This method was introduced in order to recover the components of the Tucker decomposition in equation 9. Indeed, like the standard PCA, the HOSVD method seeks to capture the maximum variation in each mode of the input tensor. Essentially, it leverages on performing SVD on matrices obtained by unfolding the tensor along each mode.

Algorithm 7 HOSVD [25]

```
Require: An order 3 tensor $\mathbf{T} \in \mathbb{R}^{d_1\times d_2\times d_3}$ and ranks $R_1, R_2, R_3$.
Output: Tucker decomposition of $\mathbf{T}$.
    Set the columns of $\boldsymbol{A}$ as the $R_1$ leading left singular vectors of $\mathrm{Mat}_1(\mathbf{T})$.
    Set the columns of $\boldsymbol{B}$ as the $R_2$ leading left singular vectors of $\mathrm{Mat}_2(\mathbf{T})$.
    Set the columns of $\boldsymbol{C}$ as the $R_3$ leading left singular vectors of $\mathrm{Mat}_3(\mathbf{T})$.
    Compute the core tensor as $\mathbf{G} \leftarrow \mathbf{T} \times_1 \boldsymbol{A}^\top \times_2 \boldsymbol{B}^\top \times_3 \boldsymbol{C}^\top$.
```

3 Applications to Machine Learning

Tensor decomposition methods find various applications in the modern machine learning paradigm. Indeed, the extraction of latent low-dimensional structures from high-dimensional arrays appears as a key step in various machine learning settings. In this section, we discuss some typical applications of tensor methods in the context of machine learning. There are several other applications that be found in the literature — for a more complete overview, see [26].

3.1 Learning Gaussian Mixtures

One of the main topics where tensor methods have proven to be successful is the problem of learning Gaussian mixtures [21]. We start by recalling the concepts of this learning problem. A Gaussian (spherical) mixture model is a probabilistic model whereby data is assumed to be generated from a mixture of k different multivariate Gaussian distributions (each representing a cluster or class) with hidden parameters or statistics (e.g. mean or covariance matrix). The density function of the spherical Gaussian mixture model is

$$p(\boldsymbol{x}) = \sum_{i=1}^{k} w_i \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_i, \sigma^2 \boldsymbol{I}_d) \qquad (17)$$

where $w_i \in [0,1]$ represents the class proportion (with $\sum_i w_i = 1$). This is generally referred to as the hidden state or latent variable, whereas the $\boldsymbol{\mu}_i \in \mathbb{R}^d$ for each $i \in [k]$ represents the means of the different clusters. For simplicity, we consider the covariances of all Gaussian components to be isotropic with variance parameter σ^2 which is further assumed to be known.

Observing n independent samples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$ following the distribution $p(\boldsymbol{x})$, the learning task aims to recover the model parameters $\{w_i, \boldsymbol{\mu}_i\}$. The classical approach to complete this task relies on maximum likelihood (ML) estimation, which is widely employed for parameter estimation. However, such a learning problem can be reformulated as a tensor decomposition problem through the method of moments [21], which interestingly does not require the Gaussianity assumption on data. Indeed, this classical approach relies on extracting the model information by computing empirical higher order moments (by averaging over the observed data samples) of the underlying data distribution. Such moments yield higher-order tensors, enabling recovery of the model parameters using tensor decomposition methods. Specifically, assuming the variance σ^2 is known, consider

$$\mathbf{T} = \mathbb{E}\left[\boldsymbol{x} \otimes \boldsymbol{x} \otimes \boldsymbol{x}\right] - \sigma^2 \sum_{j=1}^{d} \mathbf{M}_j = \sum_{i=1}^{k} w_i \boldsymbol{\mu}_i \otimes \boldsymbol{\mu}_i \otimes \boldsymbol{\mu}_i \tag{18}$$

where $\mathbf{M}_j = \mathbb{E}[\boldsymbol{x}] \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_j + \boldsymbol{e}_j \otimes \mathbb{E}[\boldsymbol{x}] \otimes \boldsymbol{e}_j + \boldsymbol{e}_j \otimes \boldsymbol{e}_j \otimes \mathbb{E}[\boldsymbol{x}]$. $\mathbf{T}$ can be approximated through the empirical moments of $\boldsymbol{x}$. These moments are given by

$$\hat{\mathbb{E}}[\boldsymbol{x}] = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i, \quad \hat{\mathbb{E}}[\boldsymbol{x} \otimes \boldsymbol{x} \otimes \boldsymbol{x}] = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i \otimes \boldsymbol{x}_i \otimes \boldsymbol{x}_i.$$

Hence, the *symmetric* tensor $\hat{\mathbf{T}}$ decomposes as

$$\hat{\mathbf{T}} = \mathbf{T} + \mathbf{E} = \sum_{i=1}^{k} w_i \boldsymbol{\mu}_i \otimes \boldsymbol{\mu}_i \otimes \boldsymbol{\mu}_i + \mathbf{E} \tag{19}$$

where $\mathbf{E}$ is a residual error tensor that vanishes if $n \to \infty$ (with d being fixed). Thus, the model parameters can be estimated by computing the best rank-k approximation of $\hat{\mathbf{T}}$.

The same concept generalizes to multi-view mixture models where multiple modalities (views) of the data are observed (text, audio, image, etc.) [25]. Specifically, these models suppose that the observed modalities are independent given some (latent) discrete categorical random variable h characterized by $\mathbb{P}(h = j) = w_j \in (0, 1)$ for $j \in [k]$ with k being the number of clusters. For simplicity, let us assume that there are three views represented by three random vectors $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}$ and $\boldsymbol{x}^{(3)}$. The multi-view mixture model supposes that the $\boldsymbol{x}^{(\ell)} \in \mathbb{R}^d$ are conditionally independent given the k-categorical latent variable $h \in [k]$. The conditional expectations for the three modalities are denoted by

$$\boldsymbol{a}_h = \mathbb{E}\left[\boldsymbol{x}^{(1)} \mid h\right], \quad \boldsymbol{b}_h = \mathbb{E}\left[\boldsymbol{x}^{(2)} \mid h\right], \quad \boldsymbol{c}_h = \mathbb{E}\left[\boldsymbol{x}^{(3)} \mid h\right].$$

Let us also assume that the random variables $\boldsymbol{x}^{(\ell)}$ are conditionally (regarding h) normally distributed. Specifically,

$$\begin{gathered}\boldsymbol{x}^{(1)} \mid h \sim \mathcal{N}(\boldsymbol{a}_h, \sigma^2 \boldsymbol{I}_d), \quad \boldsymbol{x}^{(2)} \mid h \sim \mathcal{N}(\boldsymbol{b}_h, \sigma^2 \boldsymbol{I}_d), \\ \boldsymbol{x}^{(3)} \mid h \sim \mathcal{N}(\boldsymbol{c}_h, \sigma^2 \boldsymbol{I}_d).\end{gathered} \tag{20}$$

Then, similarly to the learning Gaussian mixtures task, the multi-view model parameters $\boldsymbol{w}_i, \boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i$ can be recovered through tensor decomposition of the empirical third moment *non-symmetric* tensor $\hat{\mathbf{T}}$, the expectation of which is given by

$$\begin{aligned}\mathbf{T} &= \mathbb{E}[\boldsymbol{x}^{(1)} \otimes \boldsymbol{x}^{(2)} \otimes \boldsymbol{x}^{(3)}] = \mathbb{E}[\mathbb{E}[\boldsymbol{x}^{(1)} \otimes \boldsymbol{x}^{(2)} \otimes \boldsymbol{x}^{(3)}] \mid h] \\ &= \sum_{i=1}^{k} w_i \boldsymbol{a}_i \otimes \boldsymbol{b}_i \otimes \boldsymbol{c}_i\end{aligned}$$

Because it is possible to compute only empirical estimates of such moments, the resulting tensors are random tensors with a hidden low-rank structure — referred to as *spiked random tensors*. We will describe such tensors in more detail in Section 4.

3.2 Tensor Completion

Another challenging topic that leverages low-rank tensor representations is the tensor completion task, illustrated in Figure 7. Such a task involves a subset of a tensor's entries being observed and then recovering its missing entries. This generalizes the well-known matrix completion problem that has been widely considered in recommendation systems, e.g. the Netflix challenge [27], which involves designing a system that recommends movies for users based on a very sparse score matrix. The rows and columns of the score matrix correspond to users and movies, respectively, with its entries being the score given by each user for the watched movie. In case of multiple observed modalities (movies, songs, books, etc.), such a score matrix becomes a score *tensor* and therefore the task becomes a tensor completion problem.

The tensor completion task involves recovering the missing entries of a high dimensional tensor[3] $\mathbf{T} \in \mathbb{R}^{d \times d \times d}$ with d being large. $\mathbf{T}$ is assumed to have (approximately) low *Tucker-rank* R, specifically, it decomposes as

$$\mathbf{T} = \sum_{i=1}^{R}\sum_{j=1}^{R}\sum_{k=1}^{R} G_{ijk} \boldsymbol{a}_i \otimes \boldsymbol{b}_j \otimes \boldsymbol{c}_k \tag{21}$$

where the core tensor $\mathbf{G} \in \mathbb{R}^{R \times R \times R}$ is assumed to be cubic for convenience. We suppose that we observe n entries of $\mathbf{T}$

$$y_\ell = \mathbf{T}_{o_\ell}, \quad \ell \in [n] \tag{22}$$

where the o_ℓ's are independent and identically distributed (i.i.d.) uniformly sampled from $\{(i, j, k) : 1 \leq i, j, k \leq d\}$. Therefore, the goal is to recover the missing unobserved entries of $\mathbf{T}$ given the observed data $\{(o_i, y_i) : i \in [n]\}$.

Before discussing the tensor completion problem, we start by recalling the matrix completion formulation. Even for matrices, the natural formulation is non-convex and NP-hard. This is because it involves finding the matrix with the smallest possible rank that matches the observed data by solving the following optimization problem:

$$\min_{\boldsymbol{X} \in \mathbb{R}^{d \times d}} \operatorname{rank}(\boldsymbol{X}) \quad \text{s.t.} \quad \frac{1}{|\Omega|} \sum_{(i,j) \in \Omega} |X_{ij} - M_{ij}| \leq \varepsilon \tag{23}$$

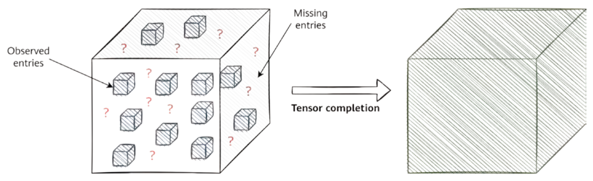


Figure 7 Illustration of the tensor completion task

[3] We assume that all the modes have the same dimension d for simplicity.

where Ω is the set of observed indices, M is the matrix we want to recover (analog of $\mathbf{T}$ above in the case of tensor completion), and ε is a fixed error tolerance parameter. The above problem admits a convex relaxation in terms of the *nuclear norm*[4] that can be solved efficiently in polynomial time. Indeed, replacing $\operatorname{rank}(\mathbf{X})$ by the nuclear norm $\|\mathbf{X}\|_*$ in the objective equation 23 yields a possible recovery of the unobserved entries provided that the set Ω is of size $n \gg Rd\log(d)$.

A basic approach to solve the tensor completion problem involves using tensor unfolding to map this problem to a matrix completion problem. The idea is to solve the following

$$\min_{\mathbf{A}\in\mathbb{R}^{d\times d\times d}} \sum_{j=1}^{3} \|\operatorname{Mat}_j(\mathbf{A})\|_* \text{ s.t. } \frac{1}{|\Omega|}\sum_{(i,j,k)\in\Omega} |A_{ijk} - T_{ijk}| \le \varepsilon$$

However, because this approach does not rely on the low-rank tensor structure, it requires a sample size of order $n \gg Rd^2 \operatorname{polylog}(d)$, which is significantly larger than the dimension of (Tucker) rank-R tensors, that is, it of order $\mathcal{O}(R^3 + Rd)$.

The nuclear norm generalizes to tensors in the following way: for a D-order tensor $\mathbf{A} \in \mathbb{R}^{d_1\times\cdots\times d_D}$, the nuclear norm of $\mathbf{A}$ is defined by [28]

$$\|\mathbf{A}\|_* = \inf_{R,\lambda_i,\boldsymbol{u}_i^{(d)}} \{\sum_{i=1}^{R}|\lambda_i| : \mathbf{A} = \sum_{i=1}^{R}\lambda_i \boldsymbol{u}_i^{(1)} \otimes\cdots\otimes \boldsymbol{u}_i^{(D)},$$
$$\|\boldsymbol{u}_i^{(k)}\| = 1, R\in\mathbb{N}\}$$

which coincides with the definition of the nuclear norm for matrices (i.e. $D = 2$). The tensor completion problem can be formulated in terms of the tensor nuclear norm as

$$\min_{\mathbf{A}\in\mathbb{R}^{d\times d\times d}} \|\mathbf{A}\|_* \text{ s.t. } \frac{1}{|\Omega|}\sum_{(i,j,k)\in\Omega} |\mathbf{A}_{ijk} - \mathbf{T}_{ijk}| \le \varepsilon \tag{24}$$

Solving the optimization problem in equation 24 ensures exact recovery of the missing entries with high probability provided that $n \gg \left(R^{\frac{1}{2}}d^{\frac{3}{2}} + R^2 d\right)\operatorname{polylog}(d)$ which improves upon the unfolding approach by order $\mathcal{O}((Rd)^{-\frac{1}{2}})$. However, tensor nuclear norm is NP hard to compute in the worst case scenario — the same is true for most tensor problems [29]. To overcome this issue, many relaxations have been proposed (e.g., Θ-norm or sum of squares [30–32]) that yield practical methods yet do not scale well for high dimensional tensors. Other techniques relying on non-convex optimization have shown to be computationally successful while approaching the sample complexity of the nuclear norm approach [33, 34].

[4] We recall that the nuclear norm $\|.\|_*$ of a matrix is the dual norm of its spectral norm and coincides with the sum of its singular values.

The tensor completion problem incorporates two sources of randomness: the underlying tensor is supposed to be *approximately* low-rank (i.e., a sum of a low-rank tensor and a random noise tensor), and the observed entries are randomly sampled.

3.3 Community Detection on Hypergraphs

The classical community detection task can be mapped to a tensor problem if data is represented as high-order networks with generalized pair-wise interactions. Such a task, essentially, involves finding a partition into *K* groups of a collection of graph data.

We start by recalling the case of simple pair-wise interactions that yield a graph defined through its adjacency matrix. The classical probabilistic model used to analyze such graphs is the so-called stochastic block model (SBM) which defines a (two communities) graph by three parameters: the number of edges n, the inter-community interaction probability p and the intra-community interaction probability q with $q < p$. In this setting, the adjacency matrix $\boldsymbol{A}$ is constructed as a random matrix with Bernoulli entries and the expectation of $\boldsymbol{A}$ has a block-wise structure as (for $n = 4$)

$$\mathbb{E}A = \begin{bmatrix} p & p & q & q \\ p & p & q & q \\ q & q & p & p \\ q & q & p & p \end{bmatrix}, v_1(\mathbb{E}A) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, v_2(\mathbb{E}A) = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

where $\boldsymbol{v}_i(\boldsymbol{M})$ is the i-th eigenvector[5] of $\boldsymbol{M}$. As such, the second eigenvector of $\mathbb{E}\boldsymbol{A}$ provides information about the partition of the graph. However, it is possible to access only the adjacency matrix $\boldsymbol{A}$ which decomposes as follows

$$\boldsymbol{A} = \underbrace{\mathbb{E}\boldsymbol{A}}_{\text{Low-rank}} + \underbrace{(\boldsymbol{A} - \mathbb{E}\boldsymbol{A})}_{\text{noise}} \tag{25}$$

$\boldsymbol{A}$ concentrates around its expectation, perturbation analysis indicates that $\boldsymbol{v}_2(\boldsymbol{A}) \approx \boldsymbol{v}_2(\mathbb{E}\boldsymbol{A})$, yielding the classical spectral method that involves the following: 1) observing $\boldsymbol{A}$, 2) computing $\boldsymbol{v}_2(\boldsymbol{A})$ and 3) using the signs of the entries of $\boldsymbol{v}_2(\boldsymbol{A})$ to recover the communities.

[5] In decreasing order of the corresponding eigenvalues.

The same ideas generalize to the concept of hypergraphs which define high-order interactions. A hypergraph $G = (V, H)$ is defined by its vertex set V and its hyperedge set H. The SBM also generalizes to hypergraphs in the following way: G is a (q-uniform) hypergraph if each hyperedge $e = \{v_1, \dots, v_q\}$ is of the same size q and appears with probability p_{out} or with p_{in} if $C(v_1) = \dots = C(v_n)$, where $C : [n] \to \{-1, 1\}$ is the community assignment function.

Observing such G, the task involves finding a label estimator $\hat{C}$ which correlates with the true assignment function C. Simple graphs can be achieved with spectral methods that rely on the *adjacency tensor* T defined as a sparse tensor of order q with n^q entries, such that $T_{i_1 \dots i_q} = 1$ if $\{i_1, \dots, i_q\}$ is a hyperedge. As in the previous examples, understanding the spectral behavior of such random tensor T is of central interest in order to quantify the performance of hypergraph spectral community detection methods. This motivates the need for random tensor theory tools to analyze such objects.

3.4 Supervised and Unsupervised Learning on Low-Rank Tensors

In addition to the applications presented earlier, many works in the literature leverage the low-rank tensor structure to design learning systems — for example tensor regression is used in a supervised setting [35], and clustering is used in an unsupervised setting [36]. The tensor structure has been shown to enhance the performance of learning models and is a key ingredient in more complex learning architectures. For example, it is employed in multi-modal data or multi-spectral images [37, 38] and in the design of advanced neural network architectures (it replaces the flattening operation in fully connected layers of a convolutional neural network with CP-based operations [39]).

As well as the performance gain shown by [39], there is a significant reduction in the number of parameters needed to describe the learned model. Indeed, the gain in the size of the parameter space can be seen when the data samples are order D tensors and have for example a rank-one underlying structure. In this case, if the tensors dimensions are $d_1 \times \cdots \times d_D$, the dimension of the parameter space can be significantly reduced from $\prod_{j=1}^{D} d_j$ to $\sum_{j=1}^{D} d_j$.

This motivates the analysis of learning algorithms when processing low-rank tensor structured data. To do so, we consider a framework where the data is assumed to be low-rank tensors perturbed by some additive noise. Based on the random tensor theory results (which will be presented in the next section), we characterize the theoretical performance of simple linear methods (in both supervised and unsupervised settings) with and without incorporating the knowledge of the low-rank structure. We show analytically that the incorporation of this knowledge enables a denoising approach that considerably improves the performance of the studied methods. In particular, performance improvements are achieved when a limited number of training samples is available or when data is of high-dimension. Exploiting the structure of the data allows us to obtain equivalent performance with far fewer samples. Such analysis relies on a specific statistical model on the data — this model is described as follows.

Statistical data model: Let the training samples be n independent tensor-structured data $\mathsf{X}_1, \dots, \mathsf{X}_n$, with each being of order D and of dimension $d_1 \times \cdots \times d_D$ (illustrated in Figure 8). We denote the dimensions $d = \sum_{j=1}^{D} d_j$ and $p = \prod_{j=1}^{D} d_j$. We assume that the X_i's are distributed in two classes $\mathcal{C}_1$ and $\mathcal{C}_2$ (of cardinality n_1 and n_2, respectively, i.e., $n = n_1 + n_2$), such that for $\mathsf{X}_i \in \mathcal{C}_a$ with $a \in \{1, 2\}$,

$$\mathsf{X}_i = (-1)^a \bigotimes_{j=1}^{D} \boldsymbol{\mu}_j + \mathsf{Z}_i \in \mathbb{R}^{d_1 \times \cdots \times d_D} \tag{26}$$

where Z_i is a random tensor with i.i.d. standard Gaussian entries, $\boldsymbol{\mu}_j \in \mathbb{R}^{d_j}$ for $j \in [D]$ are independent from the Z_i's and $\mathsf{M} = \bigotimes_{j=1}^{D} \boldsymbol{\mu}_j$ stands for the outer product between all the $\boldsymbol{\mu}_j$'s. In the context of supervised binary classification, we are further given a vector of labels $\boldsymbol{y} \in \mathbb{R}^n$ such that $y_i = -1$ for $\mathsf{X}_i \in \mathcal{C}_1$ and $y_i = 1$ for $\mathsf{X}_i \in \mathcal{C}_2$.

We denote the training data tensor $\mathsf{X} = [\mathsf{X}_1, \dots, \mathsf{X}_n] \in \mathbb{R}^{d_1 \times \cdots \times d_D \times n}$ by concatenating all the X_is along the $(D+1)$-th mode of dimension n. X is expressed in tensor form as

$$\mathsf{X} = \mathsf{M} \otimes \boldsymbol{y} + \mathsf{Z} \tag{27}$$

where $\mathsf{Z} = [\mathsf{Z}_1, \dots, \mathsf{Z}_n] \in \mathbb{R}^{d_1 \times \cdots \times d_D \times n}$. Given the rank-one structure of the tensor mean M, the outer product $\mathsf{M} \otimes \boldsymbol{y}$ results in a rank-one tensor of order $D+1$. As such, the data tensor X is a *rank-one spiked random tensor model* of order $D+1$, where the signal part is $\mathsf{M} \otimes \boldsymbol{y}$ and Z corresponds to the noise part. In order to characterize the behavior of learning methods applied on the above data model, it is necessary to understand the behavior of spiked random tensors which will be the subject of the following section.

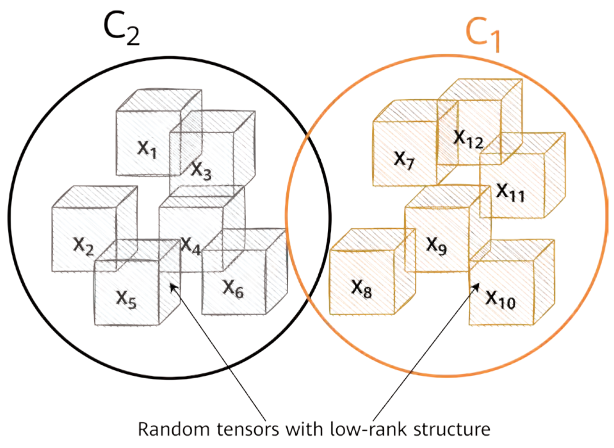


Figure 8 Illustration of the statistical data model

4 Random Tensor Theory (RTT)

As discussed in the previous section, the simplest case of rank-1 random tensors appears naturally in many machine learning problems. Although random matrix models have been extensively studied and well understood in the literature, the understanding of random tensor models is still in its infancy and the ideas from random matrix theory do not easily extend to higher-order random tensors. Therefore, analyzing higher-order random tensors requires the development of new approaches such as the one we will describe in this section.

To better illustrate and understand the effect of randomly perturbed low-rank tensor models, we consider the *asymmetric* spiked tensor model, which involves a D-order tensor $\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_D}$ of the form (illustrated in Figure 9)

$$\mathbf{T} = \beta \boldsymbol{x}_1 \otimes \cdots \otimes \boldsymbol{x}_D + \frac{1}{\sqrt{d}} \mathbf{X} \tag{29}$$

where $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_D) \in \mathbb{S}^{d_1 - 1} \times \cdots \times \mathbb{S}^{d_D - 1}$ with $\mathbb{S}^{d-1}$ denoting the unit sphere of dimension d, $\mathbf{X}$ is a random tensor with i.i.d. standard Gaussian entries $X_{i_1 \ldots i_D} \sim \mathcal{N}(0,1)$, $d = \sum_{i=1}^{D} d_i$ and $\beta \in \mathbb{R}$ is a parameter controlling the signal-to-noise ratio (SNR). Typically, observing such $\mathbf{T}$, one aims to recover the hidden rank-one information tensor (or spike) $\beta \boldsymbol{x}_1 \otimes \cdots \otimes \boldsymbol{x}_D$.

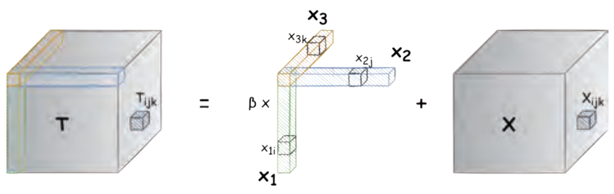


Figure 9 Illustration of the rank-one spiked random tensor of order three

Because of the noise $\mathbf{X}$, it is possible to estimate the signal component and the quality of such estimation depends on the SNR β. In this work, we are interested in studying the performance of the ML estimator of the spike given $\mathbf{T}$, which is formally defined by

$$(\lambda^*, \boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_D^*) = \underset{\lambda \in \mathbb{R}^+, \boldsymbol{u}_i \in \mathbb{R}^{d_i - 1}}{\arg\min} \|\mathbf{T} - \lambda \boldsymbol{u}_1 \otimes \cdots \otimes \boldsymbol{u}_D\|_{\mathrm{F}}^2. \tag{30}$$

Under the assumptions we describe later, it is possible to characterize the asymptotic limits of the singular value λ^* and the alignments $\langle \boldsymbol{u}_i^*, \boldsymbol{x}_i \rangle$ (which measure the degree of correlation between the estimated components $\boldsymbol{u}_i^*$ and the true spike components $\boldsymbol{x}_i$), when the dimensions of the tensor tend to infinity, specifically, $d_i \to \infty$ with dimension ratios $\frac{d_i}{d} \to c_i \in (0,1)$.

To this end, the random tensor $\mathbf{T}$ can be mapped to a symmetric random matrix $\boldsymbol{T}_D^* \in \mathbb{R}^{d \times d}$, which is constructed from contractions of $\mathbf{T}$ along $D-2$ directions with its singular vectors $\boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_D^*$ [40]. Leveraging tools from random matrix theory, the limiting distribution of the eigenvalues of $\boldsymbol{T}_D^*$ (also called spectral measure) can be characterized, and the exact expressions of the asymptotic limits of the singular value λ^* and the alignments $\langle \boldsymbol{x}_i, \boldsymbol{u}_i^* \rangle$ can be established. Indeed, the singular value and vectors λ^* and $\boldsymbol{u}_i^*$ can be characterized through a variational approach. Specifically, in equation 30, λ can be interpreted as the generalization to the tensor case of the concepts of dominant singular value, while the $\boldsymbol{u}_i$ can be interpreted as the associated singular vectors [20]. Following the variational arguments of [20], equation 30 can be reformulated using contractions of $\mathbf{T}$ as

$$\max_{\prod_{i=1}^{D} \|\boldsymbol{u}_i\| = 1} |\langle \mathbf{T}, \boldsymbol{u}_1 \otimes \cdots \otimes \boldsymbol{u}_D \rangle| \tag{31}$$

$$f(z,\beta) = z + g(z) - \beta \prod_{i=1}^{D} q_i(z,\beta), \quad q_i(z,\beta) = \left(\frac{\alpha_i(z,\beta)^{D-3}}{\prod_{j \neq i} \alpha_j(z,\beta)} \right)^{\frac{1}{2D-4}}, \quad c_i = \lim_{d_i \to \infty} \frac{d_i}{\sum_{j=1}^{D} d_j}$$
$$\alpha_i(z,\beta) = \frac{\beta}{z + g(z) - g_i(z)}, \quad g_i(z) = \frac{g(z)+z}{2} - \frac{\sqrt{4c_i + (g(z)+z)^2}}{2}, \quad g(z) = \sum_{i=1}^{D} g_i(z) \tag{28}$$

whose Lagrangian is given by $\mathbf{T}(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D) - \lambda\left(\prod_{i=1}^{D} \|\boldsymbol{u}_i\| - 1\right)$ with $\lambda > 0$. Thus, the stationary points $(\lambda, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_D)$, with each $\boldsymbol{u}_i$ being a unit vector, must satisfy the following Karush-Kuhn-Tucker conditions that generalize the conditions in equation 13 for any order D tensor, for $i \in [D]$

$$\begin{cases} \mathbf{T} \times_1 \boldsymbol{u}_1 \cdots \times_{i-1} \boldsymbol{u}_{i-1} \times_{i+1} \boldsymbol{u}_{i+1} \cdots \times_D \boldsymbol{u}_D = \lambda \boldsymbol{u}_i \\ \lambda = \mathbf{T} \times_1 \boldsymbol{u}_1 \times_2 \boldsymbol{u}_2 \cdots \times_D \boldsymbol{u}_D \end{cases} \tag{32}$$

Leveraging the identities in equation 32 which are also satisfied by $(\lambda^*, \boldsymbol{u}_1^*, \ldots, \boldsymbol{u}_D^*)$ yields the following central result. Such a result characterizes the solution of equation 30 in terms of the estimated dominant eigenvalue and the alignment of the associated estimated eigenvectors with the spike $\boldsymbol{x}_1 \otimes \cdots \otimes \boldsymbol{x}_D$.

Theorem 4.1 (See [40]). *For all $D \geq 3$, there exists $\beta_s > 0$ such that for $\beta > \beta_s$*

$$\lambda^* \xrightarrow{a.s.} \lambda^\infty, \quad |\langle \boldsymbol{u}_i^*, \boldsymbol{x}_i \rangle| \xrightarrow{a.s.} q_i(\lambda^\infty, \beta)$$

where λ^∞ satisfies $f(\lambda^\infty, \beta) = 0$ with f and q_i being defined through the deterministic fixed-point equations in equation 28.

Note that the above result is asymptotic *in the tensor dimensions* but not in the SNR. Consequently, such a result is particularly relevant in the context of practical big data problems.

4.1 Main Approach

The main approach to obtain the result in Theorem 4.1 relies on transforming the random tensor problem into a random matrix problem. This is achieved by studying an equivalent random matrix to the spiked random tensor $\mathbf{T}$. Indeed, relying on simple random matrix theory tools (e.g. Stein's lemma [41]), $\mathbf{T}$ can be associated to a random matrix that is constructed through contractions of $\mathbf{T}$ with its singular vectors. This mainly follows from equation 33, which defines such an associated random matrix that we will denote as $\boldsymbol{\Phi}_D(\mathbf{T}, \boldsymbol{u}_1^*, \cdots, \boldsymbol{u}_D^*)$ for a generic order D tensor (for its exact definition, see [40]). Figure 10 illustrates such a matrix for a third-order tensor.

The main approach involves studying this random block-wise contraction matrix, leveraging classical tools from random matrix theory. Typically, the tool used for this is Stieltjes transform, which is defined as follows.

Definition 4.2 Given some probability measure ν, the Stieltjes transform of ν is defined by $g_\nu(z) = \int \frac{d\nu(\lambda)}{\lambda - z}, z \in \mathbb{C} \setminus \mathcal{S}(\nu)$ where $\mathcal{S}(\nu)$ stands for the support of ν.

Limiting spectral measure: To demonstrate how random matrix theory applies in this context, we subsequently provide the limits of the functions we are interested in and specifically the limiting Stieltjes transform describing the distribution of the eigenvalues of $\boldsymbol{\Phi}_D(\mathbf{T}, \boldsymbol{u}_1^*, \cdots, \boldsymbol{u}_D^*)$. Indeed, exploiting some matrix algebraic identities allows us to obtain the following result, which characterizes the limiting spectral measure of the matrix $\boldsymbol{\Phi}_D(\mathbf{T}, \boldsymbol{u}_1^*, \cdots, \boldsymbol{u}_D^*)$ (i.e., the limiting distribution of its eigenvalues).

Theorem 4.3 *Assume $d_i \to \infty$ with $\frac{d_i}{\sum_j d_j} \to c_i \in (0,1)$, the empirical spectral measure of $\boldsymbol{\Phi}_D(\mathbf{T}, \boldsymbol{u}_1^*, \cdots, \boldsymbol{u}_D^*)$ converges to a deterministic measure ν whose Stieltjes transform is given by $g(z) = \sum_{i=1}^{D} g_i(z)$ such that $\Im[g(z)] > 0$ for $\Im[z] > 0$, where $\frac{1}{d} trR^{ii}(z) g_i(z) = \frac{g(z)+z}{2} - \frac{\sqrt{4c_i + (g(z)+z)^2}}{2}$.*

Theorem 4.3 expresses the limiting Stieltjes transform implicitly through a fixed-point equation. In the case where all the tensor dimensions are equal (hypercubic tensors), $g(z)$ can be expressed explicitly. Furthermore, the corresponding distribution describes a semicircle law whose density has compact support $\mathcal{S}(\nu) = \left[-2\sqrt{\frac{D-1}{D}}, 2\sqrt{\frac{D-1}{D}}\right]$. This law is expressed as

$$\nu(dx) = \frac{D}{2(D-1)\pi} \sqrt{\left(\frac{4(D-1)}{D} - x^2\right)^+}. \tag{34}$$

Figure 11 depicts the empirical spectrum of the matrix $\boldsymbol{\Phi}_3(\mathbf{T}, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3)$, where $\mathbf{T}$ is a cubic tensor $(D = 3)$ of dimension $d_i = 100$ with $\beta = 0$, and $\boldsymbol{u}_1, \boldsymbol{u}_2$ and $\boldsymbol{u}_3$ are first randomly sampled from the unit sphere and then updated with the tensor power iteration procedure in Algorithm 2. At initialization the spectrum of $\boldsymbol{\Phi}_3(\mathbf{T}, \boldsymbol{u}_1^0, \boldsymbol{u}_2^0, \boldsymbol{u}_3^0)$ converges to a semicircle law which is described by equation 34. Moreover, at each iteration t of the power method the spectrum of $\boldsymbol{\Phi}_3(\mathbf{T}, \boldsymbol{u}_1^t, \boldsymbol{u}_2^t, \boldsymbol{u}_3^t)$ keeps converging to the semicircle law while it exhibits an isolated eigenvalue due to the statistical dependency between $\mathbf{T}$ and the iterations $\boldsymbol{u}_1^t, \boldsymbol{u}_2^t, \boldsymbol{u}_3^t$. Finally, at convergence the isolated eigenvalue appears at the position $2\lambda^*$ — as depicted in Figure 11 (right) — and λ^* is asymptotically bounded by $2\sqrt{\frac{2}{3}}$ when $\beta = 0$.

By introducing some technical matrix identities it is possible to evaluate the limits of the singular value λ^* and corresponding alignments $\langle \boldsymbol{x}_i, \boldsymbol{u}_i^* \rangle$, yielding the result of Theorem 4.1. Moreover, in the case of cubic tensors (i.e., $D = 3$ and $c_i = \frac{1}{3}$), one can find explicit expressions of the asymptotic limits of the above quantities (i.e., tensor singular value and alignments) in terms of the SNR parameter β. From this, we obtain

Corollary 4.4 *If $D = 3$ and $c_i = \frac{1}{3}$, for $\beta > \beta_s = \frac{2\sqrt{3}}{3}$*

$$\lambda^* \xrightarrow{a.s.} \sqrt{\frac{\beta^2}{2} + 2 + \frac{\sqrt{3}\sqrt{(3\beta^2 - 4)^3}}{18\beta}}$$

$$|\langle \boldsymbol{x}_i, \boldsymbol{u}_i^* \rangle| \xrightarrow{a.s.} \frac{\sqrt{9\beta^2 - 12 + \frac{\sqrt{3}\sqrt{(3\beta^2-4)^3}}{\beta}} + \sqrt{9\beta^2 + 36 + \frac{\sqrt{3}\sqrt{(3\beta^2-4)^3}}{\beta}}}{6\sqrt{2}\beta}$$

Theorem 4.1 also allows us to describe the behavior of spiked random matrices. In fact, although the formulas are not defined for $D = 2$ (see expression of $q_i(z, \beta)$ in equation 28), the matrix case can be recovered by taking $D = 3$ with for example $c_3 = 0$. In such a case, the spiked tensor model becomes a spiked matrix model ($d_3 = 1$). This yields the following corollary which describes the behavior of spiked random matrices with explicit formulas for the largest singular value and the corresponding alignments.

Corollary 4.5 *Let $c \in (0, 1)$, if $D = 3$ with $c_1 = c$ and $c_2 = 1 - c$, then for $\beta > \beta_s = \sqrt[4]{c(1-c)}$ and for all $i \in \{1, 2\}$*

$$\lambda^* \xrightarrow{a.s.} \sqrt{\beta^2 + 1 + \frac{c(1-c)}{\beta^2}}, \quad |\langle \boldsymbol{x}_i, \boldsymbol{u}_i^* \rangle| \xrightarrow{a.s.} \frac{1}{\kappa(\beta, c_i)}$$

where

$$\kappa(\beta, c) = \beta \sqrt{\frac{\beta^2(\beta^2 + 1) - c(c-1)}{(\beta^4 + c(c-1))(\beta^2 + 1 - c)}}$$

while for $\beta \in [0, \beta_s]$, $\lambda^* \xrightarrow{a.s.} \sqrt{1 + 2\sqrt{c(1-c)}}$.

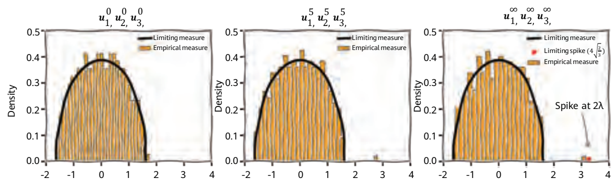


Figure 11 Spectrum of $\boldsymbol{\Phi}_3(\mathbf{T}, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3)$ at iterations $0, 5, \infty$ of tensor power iteration (see Algorithm 2) applied to $\mathbf{T}$. $d_1 = d_2 = d_3 = 100$ and $\beta = 0$. The limiting distribution is described by equation 34.

$$\begin{bmatrix} \frac{\partial u_1}{\partial X_{ijk}} \\ \frac{\partial u_2}{\partial X_{ijk}} \\ \frac{\partial u_3}{\partial X_{ijk}} \end{bmatrix} = -\frac{1}{\sqrt{d}} \left(\underbrace{\begin{bmatrix} 0_{n1\times n1} & \mathbf{T}\times_3 u_3 & \mathbf{T}\times_2 u_3 \\ (\mathbf{T}\times_3 u_3)^{\mathrm{T}} & 0_{n2\times n2} & \mathbf{T}\times_1 u_1 \\ (\mathbf{T}\times_2 u_2)^{\mathrm{T}} & (\mathbf{T}\times_1 u_1)^{\mathrm{T}} & 0_{n3\times n3} \end{bmatrix}}_{\Phi_3(\mathbf{T}, u_1, u_2, u_3)} - \lambda I_d \right)^{-1} \begin{bmatrix} u_{2j}u_{3k}\left(e_i^{d1} - u_{1i}u_1\right) \\ u_{1i}u_{3k}\left(e_j^{d2} - u_{2j}u_2\right) \\ u_{1i}u_{2j}\left(e_k^{d3} - u_{3k}u_3\right) \end{bmatrix} \in \mathbb{R}^n \quad (33)$$

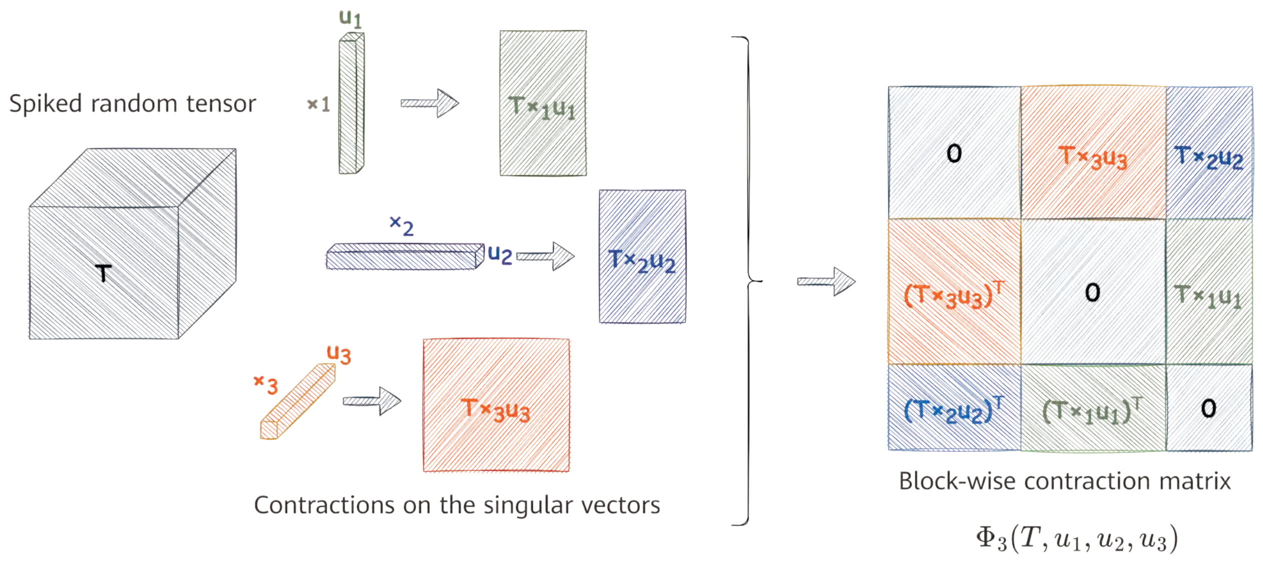

Spiked random tensor
T
×1
u1
T×1u1
×2
u2
T×2u2
×3
u3
T×3u3
Contractions on the singular vectors
0
T×3u3
T×2u2
(T×3u3)ᵀ
0
T×1u1
(T×2u2)ᵀ
(T×1u1)ᵀ
0
Block-wise contraction matrix
$\Phi_3(T, u_1, u_2, u_3)$

Figure 10 Block-wise contraction matrix $\boldsymbol{\Phi}_3(\mathbf{T}, \boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3)$ associated to a third-order tensor

4.2 Further Results on Random Tensors

Spiked random tensors: Extensive efforts have been made to study the performance of mainly rank-one tensor approximation methods in the large dimensional regime — when the tensor dimensions $d_i \to \infty$ [19, 42, 12, 43, 44, 45, 46, 40].

In particular, in the matrix case (i.e., D = 2), the spiked tensor model in equation 29 becomes a so-called spiked matrix model. For this model, in the large dimensional regime, there exists an order one critical value β_c of the SNR below which it is information-theoretically impossible to detect or recover the spike, while above β_c, it is possible to detect the spike and approximately recover the corresponding components in (at least) polynomial time using singular value decomposition (SVD). This phenomenon is sometimes known as the BBP (Baik, Ben Arous, and Péché) phase transition [47–50].

In the (symmetric) spiked tensor model for $D \geq 3$, there also exists an order one critical value[6] $\beta_c(D)$ (in the high-dimensional asymptotic) below which it is information-theoretically impossible to detect or recover the spike, while above $\beta_c(D)$ recovery is theoretically possible with the ML estimator. Computing the ML estimation in the matrix case corresponds to the computation of the largest singular vectors of the considered matrix which has a polynomial time complexity, while for $D \geq 3$, ML estimation is NP-hard [19, 51]. As such, a more practical phase transition for tensors is to characterize the algorithmic critical value $\beta_a(D, d)$ (which might depend on the tensor dimension d) above which the recovery of the spike is possible in polynomial time. Montanari and Richard [19] first introduced the symmetric spiked tensor model (of the form $\mathbf{Y} = \mu \boldsymbol{x}^{\otimes D} + \mathbf{W} \in \bigotimes^D \mathbb{R}^d$ with symmetric $\mathbf{W}$) and also considered the related algorithmic aspects. In particular, they used heuristics to highlight that spike recovery is possible, with Approximate Message Passing (AMP) or the tensor power iteration method, in polynomial time[7] provided $\mu \gtrsim d^{\frac{D-1}{2}}$. This phase transition was later proven rigorously for AMP by [12, 44] and recently for tensor power iteration by [52].

Montanari and Richard [19] further introduced a method for tensor decomposition based on tensor unfolding. This method involves unfolding $\mathbf{Y}$ to a $d \times d^{D-1}$ matrix $\mathrm{Mat}_3(\mathbf{Y}) = \mu \boldsymbol{x}\boldsymbol{y}^\top + \mathrm{Mat}_3(\mathbf{W})$, on which an SVD is then performed. They predicted that their proposed method successfully recovers the spike if $\mu \gtrsim d^{\frac{D-2}{4}}$. In a recent work by Ben Arous et al. [53], a study of spiked long rectangular random matrices[8] has been proposed under fairly general (bounded fourth-order moment) noise distribution assumptions. Therein, they proved the existence of a critical SNR for which the extreme singular value and singular vectors exhibit a BBP-type phase transition. They applied their result for the asymmetric rank-one spiked model in equation 29 (with equal dimensions) using their tensor unfolding method, and found the exact threshold obtained by [19] — specifically, $\beta \gtrsim d^{\frac{D-2}{4}}$ — for tensor unfolding to succeed in signal recovery. More recently, the authors in [40] have studied the asymmetric rank-one spiked model in equation 29 using a random matrix approach. In their work, they also described the behavior of the tensor unfolding method, yielding the same algorithmic threshold $\beta \gtrsim d^{\frac{D-2}{4}}$.

Tensor singular values and vectors: Recalling the identities in equation 32, an interesting question concerns the characterization of the stationary points (local optima or saddle points) that satisfy these identities. In particular, [54] has studied the *loss landscape* of a symmetric spiked tensor model, when its dimensions tend to infinity. The authors found that there exists $\beta_c > 0$ such that for $\beta < \beta_c$ the values taken by the objective function in equation 31 for all local maxima (including the global one) tend to concentrate in a small interval, thus the global maximum is not easily identifiable. Conversely, for $\beta > \beta_c$ the value reached by the global maximum goes out of this interval and increases with β. More recently, Goulart et al. [46] studied a *symmetric* spiked random tensor $\mathbf{Y}$ using a random matrix theory approach. They showed that there exists a threshold $0 < \beta_s < \beta_c$ such that for $\beta \in [\beta_s, \beta_c]$ there exists a local optimum of the maximum likelihood problem correlating with the spike and that such a local optimum coincides with the global optimum for $\beta > \beta_c$.

[6] Depending on the tensor order D. We will sometimes omit the dependence on D if there is no ambiguity.

[7] Using tensor power iteration or AMP with random initialization.

[8] Number of rows *m* are allowed to grow polynomially in the number of columns n (i.e., $\frac{m}{n} = n^\alpha$)

The preceding observations are conjectured to extend to *asymmetric* spiked random tensors, implying that there is a critical value — $\beta_c > 0$ — above which the maximum likelihood objective in equation 30 admits a global maximum. As for [46], the random matrix theory approach does not allow us to express such β_c. However, for asymmetric spiked random tensors, Theorem 4.1 also exhibits a threshold β_s such that for $\beta > \beta_s$, there exists a local optimum of the maximum likelihood problem correlating with the underlying signal but β_s does not coincide a priori with β_c.

4.3 Algorithmic Implications

The theoretical results from [46, 40] allow us to describe the performance of the maximum likelihood estimator (MLE) for symmetric and non-symmetric tensors respectively, providing the best rank-one approximation of $\mathbf{T}$. However, as we previously discussed, computing the MLE is NP-hard in the worst case scenario [29]. This raises the question of whether it is possible to recover the rank-one components in polynomial time. The tensor unfolding method (Algorithm 1) introduced by Ben Arous et al. [53] allows polynomial time complexity provided that the SNR $\beta \gtrsim d^{\frac{D-2}{4}}$. In turn, Theorem 4.1 allows us to provide the same algorithmic recovery threshold for the tensor unfolding method (through Corollary 4.5). The latter involves computing the dominant left singular vector of the matricized $\mathbf{T}$ (assuming that all $d_i = d/D$ are equal[9])

$$\mathrm{Mat}_i(\mathbf{T}) = \beta \boldsymbol{x}_i \boldsymbol{y}_i + \frac{1}{\sqrt{d}} \mathrm{Mat}_i(\mathbf{X}) \in \mathbb{R}^{(d/D)\times(d/D)^{D-1}}$$

where $\boldsymbol{y}_i = \mathrm{vec}(\boldsymbol{x}_1 \otimes \cdots \otimes \boldsymbol{x}_{i-1} \otimes \boldsymbol{x}_{i+1} \otimes \cdots \otimes \boldsymbol{x}_D) \in \mathbb{R}^{d^{D-1}}$. The application of Corollary 4.5 demonstrates that signal recovery is possible if the SNR β satisfies

$$\beta > D^{-\frac{D}{4}} d^{\frac{D-2}{4}} \tag{35}$$

Figure 12 depicts the asymptotic alignments and their simulated counterparts. While the tensor unfolding method has a polynomial time complexity, it does not achieve the performance of the MLE (in terms of correlation with the true signal). Conversely, while the tensor power iteration method achieves the optimal performance, it does not systematically converge (see the red dots at level 0).

[9] We use the notation $d_i = d/D$ for consistency with our previous notation $d = \sum_{i=1}^{D} d_i$.

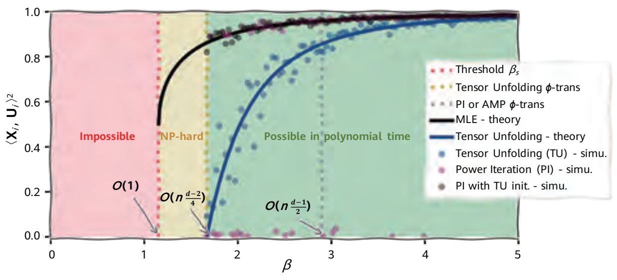


Figure 12 The different thresholds of the SNR represented in different regions. The asymptotic alignments for a cubic tensor and its unfolded version are represented in black and blue curves respectively (corresponding to Corollaries 4.4 and 4.5). Simulations are obtained using the tensor unfolding and tensor power iteration methods (Algorithms 1 and 2), and are applied to a cubic tensor of dimensions $d_i = 70$.

Interestingly, combining the two methods (tensor power iteration initialized with tensor unfolding) allows us to achieve both a better estimation threshold and the optimal correlation (see the black dots), as proved in [45].

At this stage, an open question concerns whether it is possible to design an algorithm that can recover the signal in the yellow region of Figure 12, specifically, below the algorithmic threshold $\beta > D^{-\frac{D}{4}} d^{\frac{D-2}{4}}$.

5 Application of RTT to Supervised and Unsupervised Learning

As discussed in Section 3, a large part of previous works on tensor theory being applied to machine learning problems assume a low-rank representation of input data [21, 55] and estimate this representation using the CANDECOMP/PARAFAC decomposition (CPD) [6] as the main ingredient. Indeed, the low-rank tensor structure is a sparsity hypothesis that is natural in modeling of real data seen through high-dimensional inputs [56]. However, faced with tensor-structured data, a simple and commonly used approach involves neglecting the structure and reshaping it into a set of vector samples, to which a classical machine learning algorithm is then applied. This section challenges such an approach by highlighting the fact that *a considerable gain can be obtained by taking advantage of the low-rank tensor structure of the processed data through a denoising approach, rather than treating the data as mere vectors*.

In the following sections, we analyze examples of both supervised and unsupervised learning problems based on the statistical model in equation 26 and using the random tensor tools introduced previously. In these examples, a high-dimensional regime is assumed, that is, the number of training samples n scales linearly with the tensor dimensions d_j while $\|\boldsymbol{\mu}_j\|$ remains constant.

Assumption 5.1 (Growth rate). For all $j \in [k]$, $\frac{d_j}{n} = \mathcal{O}_n(1)$ and $\|\boldsymbol{\mu}_j\| = \mathcal{O}_n(1)$[10].

Note that, in the field of random matrix theory, many classical results [57–62] assume that the feature size scales linearly with the number of samples, implying that $\prod_{j=1}^{D} d_j$ must scale linearly with n in the supposed case of tensor data. However, for $D \geq 2$, this requirement imposes a large number of training samples n which might be difficult to achieve in practical settings. As such Assumption 5.1 is more realistic from a practical perspective.

5.1 Supervised Learning

Given the training data tensor $\mathbf{X}$ in equation 27 and the corresponding labels vector $\boldsymbol{y}$, a basic learning approach [63] involves reshaping $\mathbf{X}$ into a data matrix $\mathrm{Mat}_{D+1}(\mathbf{X}) \in \mathbb{R}^{n\times p}$ with $p = \prod_{j=1}^{D} d_j$, and then learning a matched filter classifier whose parameters $\boldsymbol{w} \equiv \mathrm{vec}(\mathbf{W}) \in \mathbb{R}^p$ ($\mathbf{W} \in \mathbb{R}^{d_1\times\cdots\times d_D}$) are obtained as[11]

$$\boldsymbol{w} = \frac{1}{\sqrt{nd}} \mathrm{Mat}_{D+1}(\mathbf{X})^\top \boldsymbol{y} \tag{36}$$

where we recall that $d = \sum_{j=1}^{D} d_j$. The decision function (for a new datum $\tilde{\mathbf{X}}_i \in \mathcal{C}_a$) is given by $g(\tilde{\mathbf{X}}_i) = \langle \boldsymbol{w}, \mathrm{vec}(\tilde{\mathbf{X}}_i)\rangle$ which is equivalent in tensor notation to

$$g(\tilde{\mathbf{X}}_i) = \langle \mathbf{W}, \tilde{\mathbf{X}}_i\rangle \underset{\mathcal{C}_2}{\overset{\mathcal{C}_1}{\lessgtr}} 0, \quad \mathbf{W} = \frac{1}{\sqrt{nd}} \mathbf{X} \times_{D+1} \boldsymbol{y}. \tag{37}$$

As such, the matched filter classifier does not consider the low-rank tensor structure of the underlying data model and instead treats the data as mere vectors. Its performance is characterized by the following theorem.

Theorem 5.2 (Performance of the matched filter classifier). *Under Assumption 5.1, for $\tilde{\mathbf{X}}_i \in \mathcal{C}_a$ with $a \in \{1,2\}$ independent from the training set $\mathbf{X}$,*

$$\frac{1}{\sigma}\left(g(\tilde{\mathbf{X}}_i) - m_a\right) \xrightarrow{\mathcal{D}} \mathcal{N}(0,1)$$

where $m_a = (-1)^a \|\mathbf{M}\|^2 \sqrt{\frac{n}{d}}$ and $\sigma = \sqrt{\frac{n}{d}\|\mathbf{M}\|^2 + \frac{p}{d}}$. Moreover, the misclassification is given by $Q\left(\frac{|m_a|}{\sigma}\right)$ where Q is the Gaussian tail distribution function.

Theorem 5.2 states that the performance of the matched filter classifier depends solely on $\|\mathbf{M}\|$ and the dimension ratios $\frac{n}{d}$ and $\frac{p}{d}$. Moreover, because the data is zero-mean as per equation 26, the theoretical optimal decision threshold is $\frac{m_1+m_2}{2} = 0$. Consequently, the optimal classification is simply obtained by taking the sign of the decision function. Figure 13 provides a histogram of the decision function of the matched filter classifier and its theoretical estimate through Theorem 5.2. Under Assumption 5.1, the mean m_a remains constant while the variance σ increases due to the term $\frac{p}{d}$ as the dimension of data increases.

CP-based approach: The low-rank structure can be recovered by performing a tensor decomposition of the weights tensor $\mathbf{W}$ because this tensor is a noisy version of $\mathbf{M}$. Specifically, recalling the definition of $\mathbf{W}$ in equation 37 and $\mathbf{X}$ in equation 27, we have

$$\mathbf{W} = \sqrt{\frac{n}{d}} \bigotimes_{j=1}^{D} \boldsymbol{\mu}_j + \frac{1}{\sqrt{d}} \tilde{\mathbf{Z}} \tag{38}$$

Algorithm 8 CP-based matched filter classifier

```
Require: Tensor data $\mathbf{X}$, labels $\boldsymbol{y}$ and test datum $\tilde{\mathbf{X}}_i$.
Output: Predicted label $\tilde{y}_i = \mathrm{sign}(g_{\mathrm{CP}}(\tilde{\mathbf{X}}_i))$.
    Compute the matched filter $\mathbf{W} = \frac{1}{\sqrt{nd}} \mathbf{X} \times_{D+1} \boldsymbol{y}$
    Extract rank-1 approx of $\mathbf{W}$: $\bigotimes_{j=1}^{D} \hat{\boldsymbol{\mu}}_j = \mathrm{CPD}(\mathbf{W}, 1)$
    Set the decision function as $g_{\mathrm{CP}}(\tilde{\mathbf{X}}_i) = \langle \bigotimes_{j=1}^{D} \hat{\boldsymbol{\mu}}_j, \tilde{\mathbf{X}}_i\rangle$
```

where $\tilde{\mathbf{Z}} = \frac{1}{\sqrt{n}} \mathbf{Z} \times_{D+1} \boldsymbol{y} = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} y_i \mathbf{Z}_i$. Because $\tilde{\mathbf{Z}}$ is a sum of n i.i.d. random tensors normalized by $\sqrt{n}$, $\tilde{\mathbf{Z}}$ is also a random tensor with i.i.d. standard Gaussian entries.

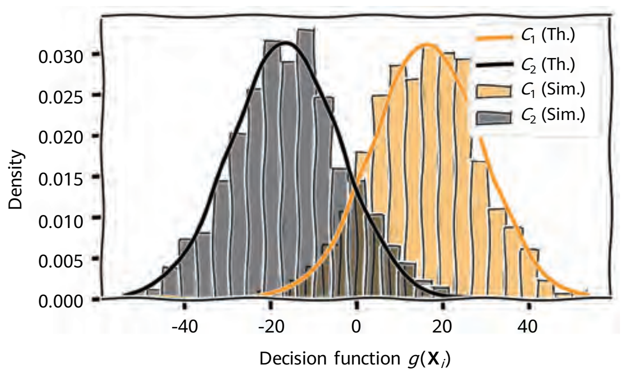


Figure 13 Theoretical versus empirical histogram of the decision function $g(\tilde{\mathbf{X}}_i)$ for the matched filter classifier as per Theorem 5.2. We considered $n = 200$ training data ($n_1 = n_2 = 100$) that are tensors of order 3 and of dimensions $d_1 = d_2 = d_3 = 20$, distributed as the rank-one tensor model in equation 26 with the $\boldsymbol{\mu}_j$'s being randomly sampled vectors from a sphere such that $\|\mathbf{M}\| = 3$.

[10] The notation $a = \mathcal{O}_n(1)$ means that a converges to a constant not depending on n if $n \to \infty$.

[11] The normalization by $\sqrt{nd}$ is adopted for convenience and does not affect the performances of the considered methods. Moreover, under Assumption 5.1, the quantities n and d are of the same order which — equivalent to the standard normalization by n.

Remark 5.3 Note that for the supervised learning setting, the Gaussianity assumption on the $\mathbf{Z}_i$ might be relaxed to any symmetric distribution with zero mean and unit variance, for which $\mathbf{Z}$ remains a random tensor with i.i.d. standard Gaussian entries by the central limit theorem.

As such, $\mathbf{W}$ is precisely a spiked random tensor model following equation 29. In order to leverage the low-rank structure of $\mathbf{W}$, we apply a rank-one CP approximation, yielding estimates of the $\boldsymbol{\mu}_j$s and then replace the weights $\mathbf{W}$ in the decision function by their rank-one approximation. This approach is outlined in Algorithm 8. In essence, extracting the rank-one component constitutes a denoising step that allows us to considerably reduce the variance of the decision function, thereby improving classification accuracy. The following result characterizes the theoretical performance of the CP-based matched filter classifier, by means of the random tensor theory results described in Section 4.

Theorem 5.4 (Performance of the CP-based matched filter classifier [64]). *Under Assumption 5.1, for $\tilde{\mathbf{X}}_i \in \mathcal{C}_a$ with $a \in \{1,2\}$ independent from the training set $\mathbf{X}$,*

$$\frac{1}{\sigma}\left(g_{\mathrm{CP}}(\tilde{\mathbf{X}}_i) - m_a\right) \xrightarrow{\mathcal{D}} \mathcal{N}(0,1)$$

where $m_a = (-1)^a \sigma \|\mathbf{M}\| \prod_{j=1}^{k} q_j\left(\sigma, \|\mathbf{M}\|\sqrt{\frac{n}{d}}\right)$ and σ satisfies $f\left(\sigma, \|\mathbf{M}\|\sqrt{\frac{n}{d}}\right) = 0$ where q_j and f are defined in equation 28. Furthermore, the misclassification error is given by $Q\left(\frac{|m_a|}{\sigma}\right)$.

Theorem 5.4 states that the performance of the CP-based matched filter classifier depends on $\|\mathbf{M}\|$ and the ratio $\frac{d}{n}$, but not on the ratio $\frac{p}{d}$ as was the case for the matched filter in Theorem 5.2. We stress that the variance σ for the CP-based classifier depends on the ratio $\frac{n}{p}$ which remains constant under Assumption 5.1. This yields a better classification accuracy compared to the naive approach where the variance scales as $\frac{p}{d}$, as we discussed previously. Indeed, Figure 14 depicts the theoretical versus empirical misclassification error for both methods, from which we observe that the CP-based matched filter classifier yields drastically better performances (almost identical to the oracle[12] which assumes perfect knowledge of $\mathbf{M}$) when n is small, or alternatively when the dimension of data is high. Indeed, Figure 16 depicts the misclassification error of both methods as a function of the ratio $\frac{n}{d}$ and $\|\mathbf{M}\|$, where we observe that the CP-based matched filter classifier performs

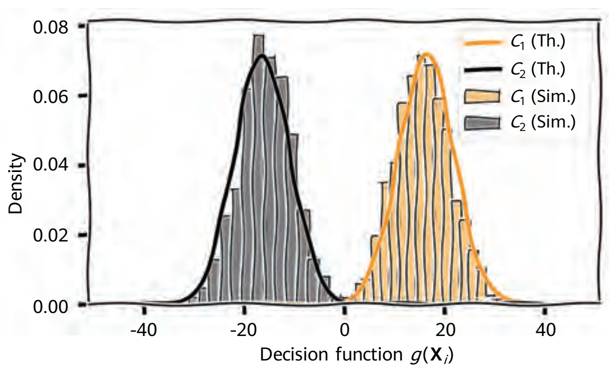


Figure 14 Theoretical versus empirical histogram of the decision function $g(\tilde{\mathbf{X}}_i)$ for the matched filter classifier as per Theorem 5.4. We considered $n = 200$ training data ($n_1 = n_2 = 100$) that are tensors of order 3 and of dimensions $d_1 = d_2 = d_3 = 20$, distributed as the rank-one tensor model in equation 26 with the $\boldsymbol{\mu}_j$'s being randomly sampled vectors from a sphere such that $\|\mathbf{M}\| = 3$.

better when $\frac{n}{d}$ is not large. This example clearly demonstrates that one can benefit from the underlying low-rank data structure, if such information is available. We will see that these conclusions also extend to an unsupervised setting, where no labels are provided.

5.2 Unsupervised Learning

In a setting where only n training samples $\mathbf{X}_1, \dots, \mathbf{X}_n$ are provided without their corresponding labels, one can rely on unsupervised learning to classify them. Given the data model in equation 27, a simple unsupervised learning approach [65] involves unfolding $\mathbf{X}$ as

$$\boldsymbol{X} = \mathrm{Mat}_{D+1}(\mathbf{X}) = \boldsymbol{y}\,\mathrm{vec}(\mathbf{M})^\top + \mathrm{Mat}_{D+1}(\mathbf{Z}) \in \mathbb{R}^{n\times p}$$

then estimating the labels $\boldsymbol{y}$ through the dominant eigenvector of the Gram matrix $\boldsymbol{X}\boldsymbol{X}^\top$ denoted by $\hat{\boldsymbol{y}}$, which coincides with the dominant left singular vector of $\boldsymbol{X}$. Therefore, Corollary 4.5 allows us to characterize the performance of this *linear spectral method*, yielding the following result.

Theorem 5.5 (Performance of linear spectral clustering). *Let $\hat{\boldsymbol{y}}$ be the right singular vector of $\boldsymbol{X}$ corresponding to its largest singular value. The estimated class for the datum $\mathbf{X}_i$ is given as $\hat{\mathcal{C}}_i = \mathrm{sign}(\hat{y}_i)$. Then under Assumption 5.1,*

$$\frac{1}{\sigma}\left(\sqrt{n}\hat{y}_i - \alpha y_i\right) \xrightarrow{\mathcal{D}} \mathcal{N}(0,1)$$

where $\alpha = \kappa\left(\|\mathbf{M}\|\sqrt{\frac{n}{p+n}}, \frac{n}{p+n}\right)^{-1}$, $\sigma = \sqrt{1-\alpha^2}$ and $\kappa(\cdot,\cdot)$ is defined in Corollary 4.5. The misclassification error is given by $Q\left(\frac{\alpha}{\sqrt{1-\alpha^2}}\right)$.

[12] The oracle classifier is defined as $\langle \mathbf{M}, \tilde{\mathbf{X}}_i \rangle$, which assumed that $\mathbf{M}$ is perfectly known.

Theorem 5.5 states that the entries of the estimated left singular vector corresponding to the largest singular value of $\boldsymbol{X}$ are Gaussian random variables, whose mean and variance depend on $\|\mathbf{M}\|$ and the ratio $c = \frac{n}{p+n}$. Essentially, in order to obtain a non-zero correlation between $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$, the signal strength $\|\mathbf{M}\|$ must be greater than $\frac{\sqrt[4]{c(1-c)}}{\sqrt{c}}$ (see Corollary 4.5). However, under Assumption 5.1, the ratio $\frac{n}{p+n} \to 0$ if $n \to \infty$, thereby yielding a high misclassification error. Figure 17 (left) depicts the $2D$ projection space corresponding to the two largest eigenvectors of $\boldsymbol{X}\boldsymbol{X}^\top$ along with its theoretical mean and fluctuations as per Theorem 5.5. In contrast, extracting the low-rank structure of the data tensor allows us to improve the classification performance. Indeed, given the data model in 27, computing a rank-1 approximation of $\mathbf{X}$ and extracting the corresponding $(D+1)$-th mode component yields an estimation of the labels vector $\boldsymbol{y}$. The following result characterizes the performance of this *CP-based clustering method*.

Theorem 5.6 (Performance of CP-based clustering [64]). *Let $\hat{\boldsymbol{y}}$ be the $(D+1)$-th mode component of the rank-1 tensor approximation of $\mathbf{X}$. The estimated class for the datum $\mathbf{X}_i$ is given as $\hat{\mathcal{C}}_i = \operatorname{sign}(\hat{y}_i)$. Then under Assumption 5.1*

$$\frac{1}{\sigma}\left(\sqrt{n}\hat{y}_i - \alpha y_i\right) \xrightarrow{\mathcal{D}} \mathcal{N}(0,1)$$

where $\alpha = q_{D+1}\left(\lambda^\infty, \|\mathbf{M}\|\sqrt{\frac{n}{d+n}}\right)$, $\sigma = \sqrt{1-\alpha^2}$ with $q_{D+1}(\cdot,\cdot)$ defined by equation 28 for a $(D+1)$-th order tensor and λ^∞ is the unique solution to $f\left(\lambda^\infty, \|\mathbf{M}\|\sqrt{\frac{n}{d+n}}\right) = 0$. Furthermore, the misclassification error is given by $Q\left(\frac{\alpha}{\sqrt{1-\alpha^2}}\right)$.

As for the linear clustering approach, the vector $\hat{\boldsymbol{y}}$ of labels

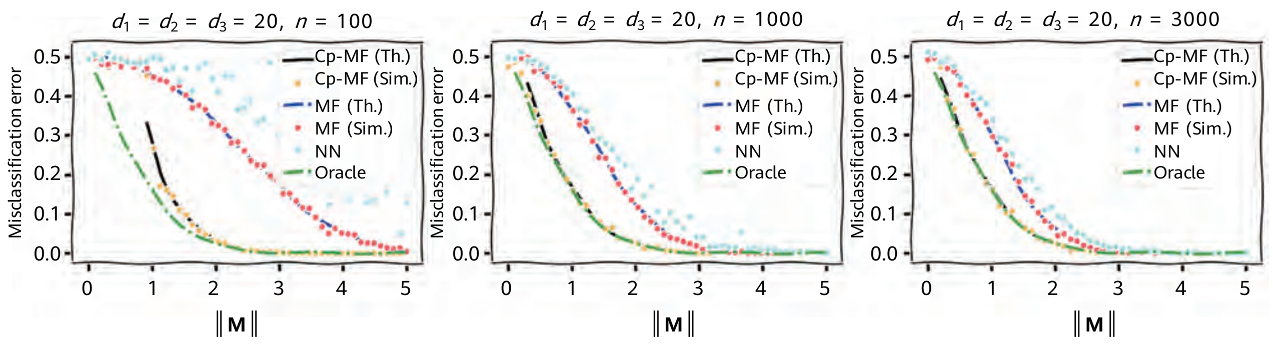


Figure 15 Theoretical versus empirical misclassification error of both matched filter (MF) and CP-based matched filter (CP-MF) classifiers. The performance of a neural network with 10 hidden neurons and ReLU activation is shown in cyan. We considered n training data as order 3 tensors of dimensions $d_1 = d_2 = d_3 = 20$ having a rank-one structure as in equation 26 with the $\boldsymbol{\mu}_j$'s being randomly sampled vectors.

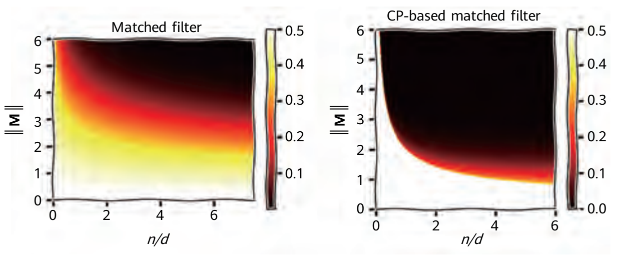


Figure 16 Theoretical misclassification error in terms of the signal strength $\|\mathbf{M}\|$ and the ratio $\frac{n}{d}$ for both MF and CP-MF as per Theorems 5.2 and 5.4 respectively

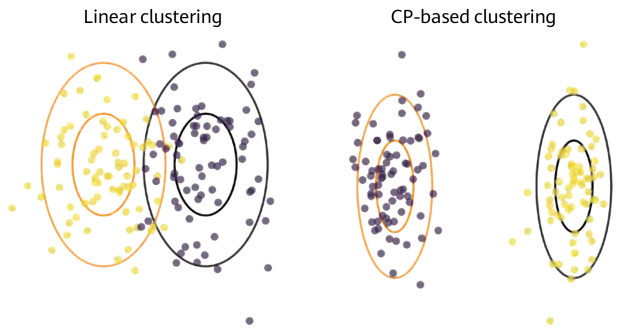


Figure 17 Left: the $2D$ projection space obtained by linear clustering. Right: the $2D$ projection space by CP-based clustering obtained through a rank-two CP decomposition of $\mathbf{X}$. We considered $D = 2$ and $n_1 = n_2 = 75$ square matrices $\mathbf{X}_i$ of size 150 generated as the model in equation 26 with $\|\mathbf{M}\| = 5$. The ellipses correspond to the theoretical means and fluctuations according to Theorems 5.5 and 5.6 respectively.

estimated with CP decomposition has Gaussian entries centered on the scaled labels $\boldsymbol{y}$ with a scaling factor α and fluctuations depending on such α. However, now the clustering performance depends on $\|\mathbf{M}\|$ and the ratio $\frac{n}{d+n}$, thereby yielding the same clustering performance as n increases and d being at least of the same order as n. Figure 17 (right) depicts the $2D$ projection space obtained by a rank-two CP decomposition of $\mathbf{X}$ with its theoretical mean and fluctuations as per Theorem 5.6. From Figure 17, we observe that denoising through computing the approximate CP decomposition yields lower variance compared to a classical linear approach, thereby allowing better clustering performance.

To best illustrate the comparison between linear clustering and CP-based clustering, let us suppose that the training data is matrices of dimension $d_1 = d_2 = n$, hence $\frac{n}{d+n} = \frac{1}{3}$. In this case, the performance of CP-based clustering is given in closed form by 4.4. Theoretically, in order to have a correlation between $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$, the signal strength $\|\mathbf{M}\|$ must be greater than 2. However, as we saw from the previous section, in order to estimate the signal in practice in polynomial time, $\|\mathbf{M}\|$ must be greater than $\frac{\sqrt[4]{c(1-c)}}{\sqrt{c}}$ with $c = \frac{1}{n+1}$, which corresponds to the phase transition of linear clustering. Figure 18 depicts the theoretical versus empirical misclassification errors along with the different thresholds for $\|\mathbf{M}\|$. This figure clearly shows the benefit of CP-based clustering upon linear clustering.

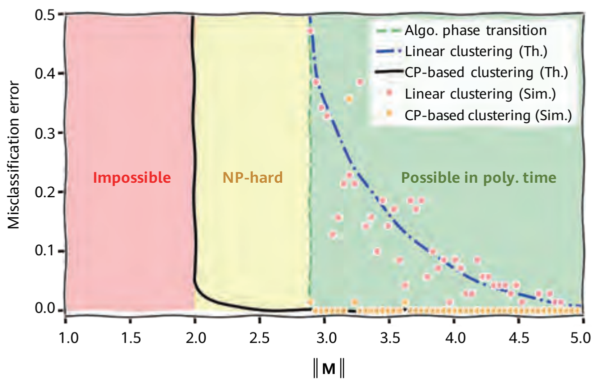


Figure 18 Theoretical versus empirical misclassification errors in terms of the signal strength $\|\mathbf{M}\|$ for both linear clustering and CP-based clustering as per Theorems 5.5 and 5.6 respectively. We considered data to be matrices such that $d_1 = d_2 = n = 70$.

6 Conclusion

This paper has presented a new and promising direction of research to understand the theoretical behavior of tensor methods, with expected impact in the field of wireless communications, signal processing, machine learning and beyond. Indeed, assessing and understanding the behavior of methods that rely on random high-dimensional tensors is of central importance in order to have theoretical guarantees about their efficiency and optimality. As illustrated in the last part of this paper, we explicitly demonstrated the application of random tensor theory to evaluate the performance of simple learning methods (such as the CP-based matched filter), whose behavior was not yet theoretically understood. This paves the way for more systematic theoretical analysis and improvement of sophisticated machine learning algorithms when dealing with tensor-structured data.

Beyond the application aspects, further work is required to set up a comprehensive theory of random tensors, by extending the developed results to more general decompositions or even to methods that rely on implicit formulations (i.e., methods that are defined through generic optimization objectives beyond the maximum likelihood formulation). There are also many open questions about the algorithmic aspects: for instance, is it possible to beat the algorithmic threshold in Figure 12 — specifically, is it possible to find a polynomial time algorithm that could recover the signal below this threshold?

References

[1] Charles Edward Spearman. General intelligence, objectively determined and measured. *American Journal of Psychology*, 15:201–293, 1904.

[2] Luca Chiantini, Giorgio Ottaviani, and Nick Vannieuwenhoven. An algorithm for generic and low-rank specific identifiability of complex tensors. *SIAM Journal on Matrix Analysis and Applications*, 35(4):1265–1287, 2014.

[3] Nicholas D. Sidiropoulos, Rasmus Bro, and Georgios B. Giannakis. Parallel factor analysis in sensor array processing. *IEEE transactions on Signal Pro-cessing*, 48(8):2377–2388, 2000.

[4] Alexis Decurninge, Ingmar Land, and Maxime Guillaud. Tensor-based modulation for unsourced massive random access. *IEEE Wireless Communications Letters*, 10(3):552–556, 2020.

[5] Pierre Comon. Tensors: a brief introduction. *IEEE Signal Processing Magazine*, 31(3):44–53, 2014.

[6] Frank L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, 1927.

[7] Richard A. Harshman et al. Foundations of the PARAFAC procedure: Models and conditions for an " explanatory" multimodal factor analysis. 1970.

[8] Carl J. Appellof and Ernest R. Davidson. Strategies for analyzing data from video fluorometric monitoring of liquid chromatographic effluents. *Analytical Chemistry*, 53(13):2053–2056, 1981.

[9] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[10] Mikael Sørensen, Ignat Domanov, and Lieven De Lathauwer. Coupled canonical polyadic decompositions and multiple shift invariance in array pro-cessing. *IEEE Transactions on Signal Processing*, 66(14):3665–3680, 2018.

[11] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[12] Thibault Lesieur, Léo Miolane, Marc Lelarge, Florent Krzakala, and Lenka Zdeborová. Statistical and computational phase transitions in spiked tensor estimation. In *Proc. IEEE International Symposium on Information Theory (ISIT)*, pages 511–515, 2017.

[13] Liqun Qi and Ziyan Luo. *Tensor analysis: spectral theory and special tensors*. SIAM, 2017.

[14] Maximilian Nickel and Volker Tresp. An analysis of tensor models for learning on structured data. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 272–287. Springer, 2013.

[15] Joseph M. Landsberg. Tensors: geometry and applications. *Representation theory*, 381(402):3, 2012.

[16] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

[17] N. Vannieuwenhoven, J. Nicaise, R. Vandebril, and K. Meerbergen. On generic nonexistence of the schmidt–eckart–young decomposition for complex tensors. *SIAM Journal on Matrix Analysis and Applications*, 35(3):886–903, 2014.

[18] Jan Draisma, Giorgio Ottaviani, and Alicia Tocino. Best rank-k approximations for tensors: generalizing eckart–young. *Research in the Mathematical Sciences*, 5(2):1–13, 2018.

[19] Andrea Montanari and Emile Richard. A statistical model for tensor PCA. *arXiv preprint arXiv:1411.1076*, 2014.

[20] Lek-Heng Lim. Singular values and eigenvalues of tensors: a variational approach. In *Proc. IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing*, pages 129–132, 2005.

[21] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of machine learning research*, 15:2773–2832, 2014.

[22] Shmuel Friedland and Venu Tammali. Low-rank approximation of tensors. In *Numerical algebra,*

matrix theory, differential-algebraic equations and control theory, pages 377–411. Springer, 2015.

[23] Alex P. da Silva, Pierre Comon, and André L.F. de Almeida. An iterative deflation algorithm for exact CP tensor decomposition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3961–3965. IEEE, 2015.

[24] Sue E Leurgans, Robert T Ross, and Rebecca B Abel. A decomposition for three-way arrays. *SIAM Journal on Matrix Analysis and Applications*, 14(4):1064–1083, 1993.

[25] Stephan Rabanser, Oleksandr Shchur, and Stephan Günnemann. Introduction to tensor decompositions and their applications in machine learning. *arXiv preprint arXiv:1711.10781*, 2017.

[26] Will Wei Sun, Botao Hao, and Lexin Li. Tensors in modern statistical learning. *Wiley StatsRef: Statistics Reference Online*, pages 1–25, 2014.

[27] Andrey Feuerverger, Yu He, and Shashi Khatri. Statistical significance of the netflix challenge. *Statistical Science*, 27(2):202–231, 2012.

[28] Shmuel Friedland and Lek-Heng Lim. Nuclear norm of higher-order tensors. *Mathematics of Computation*, 87(311):1255–1281, 2018.

[29] Christopher J Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):1–39, 2013.

[30] Carsten W Scherer and Camile WJ Hol. Matrix sum-of-squares relaxations for robust semi-definite programs. *Mathematical programming*, 107(1):189–211, 2006.

[31] Holger Rauhut and Željka Stojanac. Tensor theta norms and low rank recovery. *arXiv preprint arXiv:1505.05175*, 2015.

[32] Tengyu Ma, Jonathan Shi, and David Steurer. Polynomial-time tensor decompositions with sum-of-squares. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 438–446. IEEE, 2016.

[33] Changxiao Cai, Gen Li, H. Vincent Poor, and Yuxin Chen. Nonconvex low-rank tensor completion from noisy data. *Advances in neural information processing systems*, 32, 2019.

[34] Allen Liu and Ankur Moitra. Tensor completion made practical. *Advances in Neural Information Processing Systems*, 33:18905–18916, 2020.

[35] H. Zhou, L. Li, and H. Zhu. Tensor regression with applications in neuroimaging data analysis. *Journal of the American Statistical Association*, 108, 2013.

[36] W. Sun and L. Li. Dynamic tensor clustering. *Journal of the American Statistical Association*, 114:1894–1907, 2019.

[37] Paul Pu Liang, Zhun Liu, Yao-Hung Hubert Tsai, Qibin Zhao, Ruslan Salakhutdinov, and Louis-Philippe Morency. Learning representations from imperfect time series data via tensor rank regularization. *arXiv preprint arXiv:1907.01011*, 2019.

[38] Wanli Chen, Xinge Zhu, Ruoqi Sun, Junjun He, Ruiyu Li, Xiaoyong Shen, and Bei Yu. Tensor low-rank reconstruction for semantic segmentation. In *European Conference on Computer Vision*, pages 52–69. Springer, 2020.

[39] Jean Kossaifi, Zachary C Lipton, Arinbjörn Kolbeinsson, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor regression networks. *Journal of Machine Learning Research*, 21:1–21, 2020.

[40] Mohamed El Amine Seddik, Maxime Guillaud, and Romain Couillet. When random tensors meet random matrices. *arXiv preprint arXiv:2112.12348*, 2021.

[41] Charles M. Stein. Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, pages 1135–1151, 1981.

[42] Amelia Perry, Alexander S. Wein, and Afonso S. Bandeira. Statistical limits of spiked tensor models. In *Annales de l'Institut Henri Poincaré, Probabilités et Statistiques*, volume 56, pages 230–264. Institut Henri Poincaré, 2020.

[43] Madeline Curtis Handschy. *Phase Transition in Random Tensors with Multiple Spikes*. PhD thesis, University of Minnesota, 2019.

[44] Aukosh Jagannath, Patrick Lopatto, and Leo Miolane. Statistical thresholds for tensor PCA. *The Annals of Applied Probability*, 30(4):1910–1933, 2020.

[45] Arnab Auddy and Ming Yuan. On estimating rank-one spiked tensors in the presence of heavy tailed errors. *arXiv preprint arXiv:2107.09660*, 2021.

[46] José Henrique Goulart, Romain Couillet, and Pierre Comon. A random matrix perspective on random tensors. *stat*, 1050:2, 2021.

[47] Jinho Baik, Gérard Ben Arous, and Sandrine Péché. Phase transition of the largest eigenvalue for nonnull complex sample covariance matrices. *The Annals of Probability*, 33(5):1643–1697, 2005.

[48] Florent Benaych-Georges and Raj Rao Nadakuditi. The eigenvalues and eigenvectors of finite, low rank perturbations of large random matrices. *Advances in Mathematics*, 227(1):494–521, 2011.

[49] Mireille Capitaine, Catherine Donati-Martin, and Delphine Féral. The largest eigenvalues of finite rank deformation of large wigner matrices: convergence and nonuniversality of the fluctuations. *The Annals of Probability*, 37(1):1–47, 2009.

[50] Sandrine Péché. The largest eigenvalue of small rank perturbations of hermitian random matrices. *Probability Theory and Related Fields*, 134(1):127–173, 2006.

[51] Giulio Biroli, Chiara Cammarota, and Federico Ricci-Tersenghi. How to iron out rough landscapes and get optimal performances: averaged gradient descent and its application to tensor pca. *Journal of Physics A: Mathematical and Theoretical*, 53(17):174003, 2020.

[52] Jiaoyang Huang, Daniel Z Huang, Qing Yang, and Guang Cheng. Power iteration for tensor pca. *arXiv preprint arXiv:2012.13669*, 2020.

[53] Gérard Ben Arous, Daniel Zhengyu Huang, and Jiaoyang Huang. Long random matrices and tensor unfolding. *arXiv preprint arXiv:2110.10210*, 2021.

[54] Gerard Ben Arous, Song Mei, Andrea Montanari, and Mihai Nica. The landscape of the spiked tensor model. *Communications on Pure and Applied Mathematics*, 72(11):2282–2330, 2019.

[55] Jonathan Kadmon and Surya Ganguli. Statistical mechanics of low-rank tensor decomposition. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124016, 2019.

[56] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51:455–500, 2019.

[57] Jeffrey Pennington and Pratik Worah. Nonlinear random matrix theory for deep learning. 2017.

[58] Cosme Louart, Zhenyu Liao, and Romain Couillet. A random matrix approach to neural networks. *The Annals of Applied Probability*, 28(2):1190–1248, 2018.

[59] Hafiz Tiomoko Ali and Romain Couillet. Improved spectral community detection in large heterogeneous networks. *The Journal of Machine Learning Research*, 18(1):8344–8392, 2017.

[60] Xiaoyi Mai and Romain Couillet. A random matrix analysis and improvement of semi-supervised learning for large dimensional data. *The Journal of Machine Learning Research*, 19(1):3074–3100, 2018.

[61] Malik Tiomoko, Romain Couillet, and Hafiz Tiomoko. Large dimensional analysis and improvement of multi task learning. *arXiv preprint arXiv:2009.01591*, 2020.

[62] Mohamed El Amine Seddik, Cosme Louart, Romain Couillet, and Mohamed Tamaazousti. The unexpected deterministic and universal behavior of large softmax classifiers. In *Proc. International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1045–1053, 2021.

[63] Malik Tiomoko, Romain Couillet, and Frédéric Pascal. Pca-based multi task learning: a random matrix approach. *arXiv preprint arXiv:2111.00924*, 2021.

[64] Mohamed El Amine Seddik, Alexis Decurninge, Malik Tiomoko, Maxime Guillaud, and Romain Couillet. On learning from tensor-structured data: A random tensor theory approach. In *Submitted to ICML*, 2022.

[65] Andrew Y Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856, 2002.

Dynamical Systems and Control Theory Perspective of Computation

Jie Sun *, Daniel Ebler, Leonarduzzi Roberto Fabio
Huawei Hong Kong Research Center

Abstract

We present an alternative view of computation based on dynamical systems and control theory. In particular, we formulate computation as a problem of driving a dynamical system from its initial state to some final state that represents the target output. In this sense, a physical computation system can be interpreted as a control system, where external energy is inserted to ensure that the system's dynamics evolve toward its desired target: the output of the computation. In digital computation, this control is typically employed at every logic gate to ensure the highest level of accuracy per Boolean step. On the other hand, in analog computation, the amount and frequency of control is less demanding, thus requiring less physical energy but at the expense of higher error rates per single step. Neither takes full advantage of the nature of the physical systems underlying computation. We offer some perspectives from dynamical systems and control theory, which shed light on the potential development of a new theoretical framework for designing hybrid physical systems that can achieve optimal trade-off between energy consumption and accuracy, paving the way for achieving significantly reduced energy cost for large-scale computation.

* Corresponding author

1 Introduction

"Computers are physical systems: the laws of physics dictate what they can and cannot do. In particular, the speed with which a physical device can process information is limited by its energy and the amount of information that it can process is limited by the number of degrees of freedom it possesses."

Seth Lloyd, in "Ultimate physical limits to computation" Nature **406**, pp. 1047-1054 (2000).

A fundamental breakthrough which enables modern, large-scale computing is through the realization that, conceptually, a computational task can be decomposed into a series of basic operations, for example, Boolean (logical) operations defined on 0's and 1's [1, 2]. Under this framework, and taking into account the fact that computation of any sort relies upon physical processes, computational speedup can be generally achieved from two distinct lines of research. One line of research concerns improving the physical process that realizes the basic operations, typically the logic gates [3]. The other line of research focuses on designing efficient algorithms, that is, constructing some appropriate series of operations that achieve a prescribed function [4]. From this point of view, the optimization of computational efficiency can be clearly separated into two stages: a physical stage concerned with physical implementation, and a mathematical/logic stage concerned with the logic combination of gates to achieve a particular functional dependence. In fact, each of those two lines has formed its own research field, the former leading to integrated circuits and the latter corresponding to algorithm research [5].

We ask, what defines computation? The mainstream viewpoint interprets the process of computation as a sequence of operations to transform an input to some desired output. This perspective, while valid, restricts the discussion of computation only to its logical layer, leaving the physical process underlying the basic operations intact. The purpose of this article is to provide an alternative viewpoint of computation with an attempt to bring in the physical process together with the logical operations for joint consideration. Interestingly, we found such joint modeling corresponds to a well-established field of research in mathematics, called *dynamical systems* which typically arise in the modeling of physical processes. Figure 1 provides a visual illustration to compare the classical versus dynamics-based view of computation. From this perspective, we propose that the physical energy required for carrying out a particular computation can be decomposed into two parts, one contributing to driving the dynamics of the physical process and the other serving for control purposes to ensure self-correction of the dynamical trajectory. This viewpoint offers an opportunity to reconcile the seemingly distinct types of computation, namely digital computation and analog computation. Both are dynamical systems subject to control.

2 Dynamical Systems Viewpoint of Computation

Instead of analyzing the intermediate steps of computation, we focus on the purpose of computation, which generally means to achieve some particular end-to-end transformation from input to output [6]. Mathematically, we interpret it as a mapping Q from some element x in the input space $\mathcal{X}$ to an element y in the output space $\mathcal{Y}$,

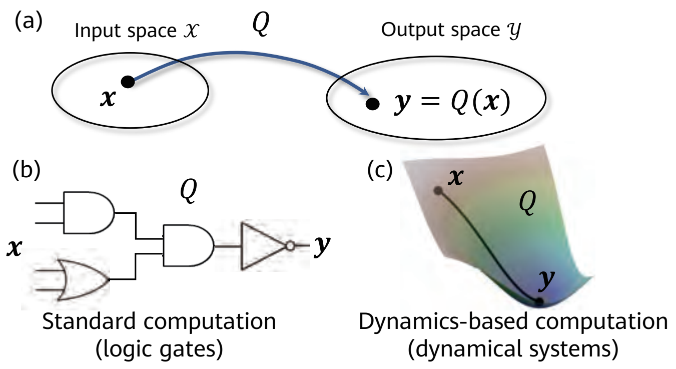

(a)
Input space $\mathcal{X}$
Q
Output space $\mathcal{Y}$
x
$y = Q(x)$
(b)
Q
x
y
Standard computation (logic gates)
(c)
x
Q
y
Dynamics-based computation (dynamical systems)

Figure 1 Dynamical system view of computation. (a) Computation is interpreted as a mapping Q from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$. (b) In the standard model of computation, the mapping Q is implemented by a circuit of logic gates and the software that runs on it. (c) In the view of computation based on dynamical systems, the mapping Q is implemented through the landscape of a dynamical system; inputs converge to the stable states of the system, which represent the outputs of computation.

$$\boldsymbol{y} = Q(\boldsymbol{x}). \tag{1}$$

For instance, given the coefficients $(a,b) \in \mathbb{R}^2$ that define a linear equation $az + b = 0$, its solution can be computed as $z = -b/a$ as long as $a \neq 0$. In this particular case, $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$ and $\boldsymbol{y} = Q(\boldsymbol{x}) = -x_2/x_1$. As an alternative example, suppose that $f(x;\boldsymbol{\mu}) = (x-\mu_1)^2 + \mu_2$ which is completely specified by the parameter vector $\boldsymbol{\mu} = (\mu_1, \mu_2)$, and we wish to *compute* the value of x for which $f(x)$ is minimized — obviously this is when $x = \mu_1$, that is, for this example $y = Q(\boldsymbol{\mu}) = \mu_1$. In this second case, the form of computation is representative of a *projection* or some type of targeted *embedding* from the input space to a lower dimensional manifold. In general we assume that the desired mapping, i.e., the ideal output of the computation, is deterministic — extension to random variables would require additional set of theoretical tools from probability theory and functional analysis.

Because most computational platforms and devices rely essentially on physical processes, it is useful to introduce some standard mathematical tools. Typically, a physical process can be represented by a time-evolving system characterized by states whose dynamics follow certain physical laws [7],

$$\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}}{dt} = f(\boldsymbol{x}; \boldsymbol{u}(t)), \tag{2}$$

where $\boldsymbol{x} \in \mathcal{D}$ represents the state of the system, f models the evolution dynamics, and $\boldsymbol{u}(t)$ denotes external control which enables modification of the system dynamics. In the absence of control, we simply write the right-hand side of the equation as $f(\boldsymbol{x})$ and denote the solution trajectory as $\phi(t;\boldsymbol{x}_0)$, which satisfies the conditions

$$\begin{cases} \phi(0;\boldsymbol{x}) = \boldsymbol{x}_0, \\ d\phi(t;\boldsymbol{x})/dt = f(\phi(t;\boldsymbol{x})). \end{cases} \tag{3}$$

This particular mapping ϕ is often referred to as the *flow* of the dynamical system [7]. To utilize a dynamical system for computation, control is generally required, except for very rare cases where the natural dynamics spontaneously evolve toward its final state that coincides with the desired target of computation. Indeed, even for those cases where the dynamical system is carefully chosen or designed to correspond to an exact computation, several factors including noise and multistability often render the flow of the system away from the ideal trajectory. In either case, a control scheme is typically needed to drive the system toward its desired output.

Example. As a concrete example, let us consider a commonly encountered problem in large-scale computation, namely the determination of the dominant eigenvector of a matrix [8], where we denote the matrix as $G \in \mathbb{R}^{n\times n}$. Such problems arise naturally in a number of important applications including search engines [9], partitioning graphs and networks [10], as well as principal component analysis [11]. For simplicity of discussion, let us assume that G is (1) symmetric, (2) admits an eigenvalue decomposition: $G = P\Lambda P^{-1}$ and (3) contains a non-degenerate maximum eigenvalue denoted as $\lambda_1 > 0$ whereas the rest of the eigenvalues satisfy $|\lambda_i| < \lambda_1$ $(\forall i = 2, 3, \ldots, n)$. Denote the corresponding (normalized) eigenvectors as $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n\}$, defining the matrix $P = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n]$. Suppose that we are given this matrix G as input and wish to *compute* its dominant eigenvector $\boldsymbol{v}_1(G)$ corresponding to λ_1. In our notation, we would write $Q(G) = \boldsymbol{v}_1(G)$. This is a standard eigenvector computation problem, with many alternative methods of computation [12]. Here we show that this task of computation can also be formulated using dynamical systems following the recipes described in [13] by attempting to embed the target output of computation as a stable state of a controlled dynamical system. One possibility is to construct a linear time-varying system with control imposed to adjust the overall magnitude of the state, as follows,

$$d\boldsymbol{x}/dt = u(t)G\boldsymbol{x}. \tag{4}$$

The idea is that the operator G, when applied iteratively, keeps driving the state of the system toward a direction aligned with $\boldsymbol{v}_1$ but its magnitude can become unbounded (when $|\lambda_1| > 1$) or diminishing (when $|\lambda_1| < 1$) and thus requires external control. By the change of basis: $\boldsymbol{y} = P^{-1}\boldsymbol{x}$, it follows that the solution of the dynamical system (4) is given by

$$y_i(t) = y_i(0)e^{\lambda_i t}e^{-\int_0^t u(\tau)d\tau}. \tag{5}$$

We can design a control input as follows,

$$u(t) = \frac{1}{\|x(t)\|^2}\sum_i x_i(t)\dot{x}_i(t) \tag{6}$$

from which we can show that the solution of the dynamical system satisfies $\boldsymbol{x} \to \boldsymbol{v}_1$ as $t \to \infty$, thus producing the desired output.

Note that in this example, no explicit Boolean logic is involved, nor is there need to choose a precise time to end the dynamics. Virtually all is needed is a (physical) process that implements a simple linear differential equation with some scalar feedback control. In fact, one can design more sophisticated coupled systems from the basic equations to solve for the other eigenvectors as well.

Interestingly, the above example, simple as it seems, encompasses the fundamental equations of motion for (both classical and quantum) physical systems. For instance, the dynamical equation $d\boldsymbol{x}/dt = cH(t)\boldsymbol{x}$ yields the Schrödinger equation for the choice of constant $c = -i\hbar$, where $H(t)$ is the Hamiltonian of the system. In such scenario, Equation (4) would amount to $u(t)G = H(t)$ as a time dependent Hamiltonian of the physical system describing the interaction and drive of the dynamics. [14, 15]

3 Multistability as a Source of Computational Capacity

Physical systems, despite their complexity, often exhibit low-dimensional behavior [16]. For example, an unforced simple pendulum oscillates periodically around its equilibrium position. For the purpose of computation, we are usually interested in and concerned with the *asymptotic* behavior of the system. Richness of the asymptotic states then directly determines the level of flexibility, or in some sense the *capacity* when such a system is used for computation. In dynamical systems, the phenomenon that a system can, under different initial conditions and perturbations, evolve toward different final stable states is called *multistability* [17].

Example. To demonstrate the notion of multistability [18, 19], let us consider a simple yet concrete example, where the dynamics of the system are *bistable* in the absence of control, and denote by $\mathcal{D}_0$ and $\mathcal{D}_1$ the set of initial states from which the system evolves toward the corresponding two stable states s_0 and s_1, that is,

$$\mathcal{D}_i = \{\boldsymbol{x_0} | \phi(t; \boldsymbol{x_0}) \to s_i \ (t \to \infty)\}. \tag{7}$$

Suppose that the system is at a particular state $\boldsymbol{x}_0 \in \mathcal{D}_0$ but we wish instead for the system to evolve toward s_1. One way to accomplish this is by inserting an impulse control, $\boldsymbol{u}_{01} \in \{\boldsymbol{u} | \boldsymbol{u} + \boldsymbol{x}_0 \in \mathcal{D}_1\}$. We demonstrate this by a simple toy example, with a one-dimensional system [20]

$$\dot{x} = dx/dt = f(x) = x(1 - x^2), \tag{8}$$

with two stable states $(s_0, s_1) = (-1, 1)$ and corresponding basins of attraction $\mathcal{D}_0 = (-\infty, 0)$ and $\mathcal{D}_1 = (0, \infty)$. If the system initiates from the state x_0, by inserting a positive (negative) control signal to alter the initial state, the system evolves toward $1(-1)$ which can be used to represent a binary output, encoding $1(0)$, as illustrated in Figure 2 (a–b). Furthermore, by using two such systems, one obtains a pair of binary outputs which can then be used to represent any pairwise Boolean operation (AND, OR, XOR, etc.).

Next, we demonstrate how one can couple several such continuous dynamics into a more complex system that is capable of information processing and computation [21, 22]. In particular, consider a coupled system with n units where the dynamics of the i-th unit follows the equation

$$\dot{x}_i = F(\boldsymbol{x}) = f(x_i) + \kappa \sum_j [f(x_j) - f(x_i)], \tag{9}$$

where $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and $f(x) = x(1 - x^2)$ as in Equation (8). Since $f(\pm 1) = 0$, it follows that each state $\boldsymbol{x}$ for which $|x_i| = 1 (\forall i)$ is an equilibrium state of the system. For $\kappa \ll 1$, we can show that each such state is locally

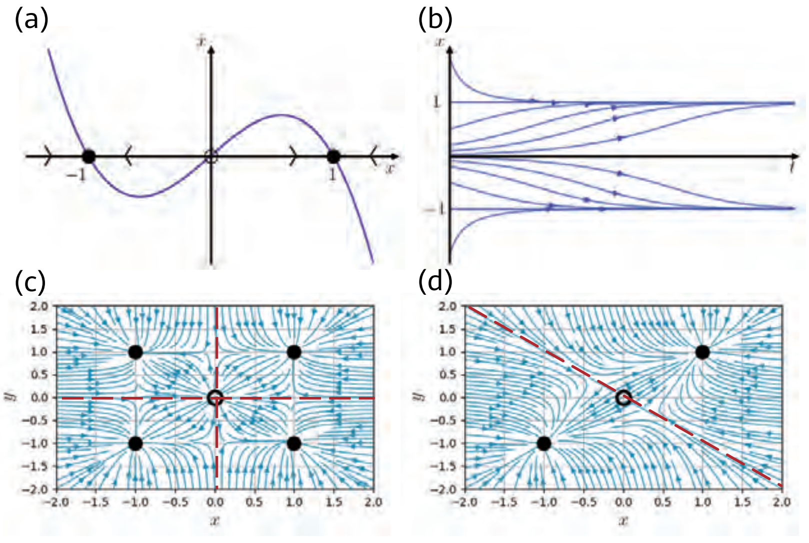


Figure 2 Examples of dynamical systems. (a) Phase portrait for the dynamical system in Equation (8). (b) Solution curves for different initial conditions; the solutions converge to ± 1 depending on the initial condition. (c–d) Phase portraits for a 2D dynamical system as in Equation (11) with $f(x) = x(1 - x^2)$, for a weak coupling $\kappa = 0.05$ (c) and a strong coupling $\kappa = 0.5$ (d); the boundaries of the basins of attraction are shown in red dashed lines.

stable. Thus, for small values of κ, the coupled dynamical system (9) contains at least 2^n stable states of the set

$$S_n = \{\boldsymbol{x} = (x_1, \dots, x_n) | x_i = \pm 1,\ \forall i\} = \{\boldsymbol{s}_1, \boldsymbol{s}_2, \dots, \boldsymbol{s}_{2^n}\}. \quad (10)$$

For $\kappa = 0$, the system is uncoupled with n isolated units which need to be individually controlled — in fact, to reach any particular final state $(x_1, \dots, x_n)$ with $|x_i| = 1$, it is necessary to individually control the state of each variable to either positive or negative based on $\text{sign}(x_i)$. On the other hand, when $\kappa \neq 0$, the system is coupled and even if the equilibrium states are the same their corresponding basins of attraction can become more complicated. The behavior of the coupled system is illustrated in Figure 2 (c–d) for different coupling strengths.

The coupling in the system can be made more general, for example, by imposing a network structure represented by some matrix $G = [G_{ij}]$, thus extending Equation (9) to a network coupled system [23]

$$\dot{x}_i = F(\boldsymbol{x}) = f(x_i) + \kappa \sum_j G_{ij} \left[f(x_j) - f(x_i) \right], \quad (11)$$

where, in addition to the effect of coupling strength κ, the coupling matrix G allows one to modify the symmetry in the system — effectively biasing its dynamics toward some of the equilibrium states. In principle, by choosing G appropriately one can achieve larger basins of attraction than the uncoupled case, making it easier for the system to evolve toward a selected set of final states [24], thus improving the overall efficiency of computation if those states are representative of the desired computational results.

In conclusion, coupling a system's states can generally lead to increased capacity since the existence of several multistable states allows a system to represent multiple solutions. In general, let us denote the basin of attraction of each stable state $\boldsymbol{s}_k \in S_n$ as

$$\mathcal{B}(\boldsymbol{s}_k) = \{\boldsymbol{x_0} | \phi(t; \boldsymbol{x_0}) \to \boldsymbol{s}_k \ (t \to \infty)\}. \quad (12)$$

The final solution found by a system will depend on these basins of attraction, whose shapes can be changed through the coupling matrix G.

For a given input $\boldsymbol{x}$, the desired computation output $Q(\boldsymbol{x})$ can be obtained from the stable state $\boldsymbol{s}_k$ to which the system converges. For example, if the goal of computation is to *classify* the inputs, then this can be achieved by assigning each of the multistable states a class label. Then, instead of designing some sophisticated Boolean logic, all we need is to encode the data into the input of the dynamical system, wait for transient dynamics to wash out, and finally read off the output based on which state the system converged to. In fact, such computation is even noise-resilient to some extent, because as long as the trajectory of the system stays within its original basin of attraction, the classification output is always going to be correct [25]. Such ideas have led to an interesting venue of research called *reservoir computing* [26], where rich information of the desired computation is embedded in a pre-wired (physical) dynamical system called a reservoir leaving a few linear parameters that are trained/adjusted to meet particular computation requirements.

4 Optimal Control Perspective of the (Energy) Cost of Computation

"All stable processes we shall predict. All unstable processes we shall control."

John von Neumann

Given plenty of examples where a dynamical system can serve as a computing device where control is used to drive the system from its initial state to a final state encoding the target output of computation [27, 28, 29, 30, 22], it is natural to ask what might be a good or even optimal way of setting the control input? Acknowledging that the particular objective of control is itself challenging to uniquely define due sometimes to unavoidable gaps between theoretical and numerical measures of controllability and control trajectories [31], we argue that at least conceptually the question above can be formulated mathematically as an optimal control problem [32, 33], where the "best" $\boldsymbol{u}(t)$ is sought to minimize a cost functional J. Assume that the computation runs for a time interval $t_f > 0$ (when no constraint is imposed we set $t_f = \infty$). Our cost functional J will be composed of two elements: first, an *endpoint cost* S associated to the final state of the dynamical system; and second, a *running cost* that measures the control cost incurred during each step of the computation (measured by a functional V). Then, the "best" $\boldsymbol{u}(t)$ is sought to minimize the functional

$$J = S(\boldsymbol{x}(t_f), t_f) + \int_0^{t_f} V(\boldsymbol{x}(t), \boldsymbol{u}(t)) dt, \quad (13)$$

where S and V are cost functions which jointly determine the objective of control optimization and "cost" here reflects the total control efforts incurred during the entire time interval.

From this viewpoint, we interpret computation as a control problem by identifying the input of the computation as $\boldsymbol{x}_0$ and thus the desired output becomes $Q(\boldsymbol{x}_0)$. First, we choose an endpoint cost that quantifies how far the final state is from the desired solution, for instance by letting $S(\boldsymbol{x}(t_f), t_f) = \|Q(\boldsymbol{x}_0) - \boldsymbol{x}(t_f)\|^2$. Then, we select a running cost that measures the energy spent by the control function, one example being $V(\boldsymbol{x}(t), \boldsymbol{u}(t)) = \alpha \boldsymbol{u}(t)^\top \boldsymbol{u}(t)$. Then we arrive at the optimal control problem

$$\begin{cases} \min_{\boldsymbol{u}} J[\boldsymbol{u}] = \|Q(\boldsymbol{x}_0) - \boldsymbol{x}(t_f)\|^2 + \alpha \int_0^{t_f} \boldsymbol{u}(t)^\top \boldsymbol{u}(t) dt, \\ \text{subject to: } d\boldsymbol{x}/dt = f(\boldsymbol{x}; \boldsymbol{u}(t)), \ \boldsymbol{x}(0) = \boldsymbol{x}_0. \end{cases} \tag{14}$$

The goal is to design an appropriate control input $\boldsymbol{u}(t)$ such that the state of the system evolves toward the desired computation output, $Q(\boldsymbol{x}_0)$, with minimal cost of control. Here the parameter α serves to achieve a trade-off between controlling (computation) accuracy and controlling cost, similar to the so-called regularization parameter often encountered in learning theory [34]: for $\alpha \ll 1$, the focus is placed on ensuring the final state of the system reaching the target $Q(\boldsymbol{x}_0)$, (almost) ignoring the cost of control; on the other hand, for $\alpha \gg 1$, little emphasis is given to the state of the system but more on ensuring that the control input is minimal.

5 Control as a Form of Error-Correction

Any realistic physical process is subject to noise and perturbations, thus transforming into a non-zero probability of error, as represented by the system evolving toward a state that differs from its desired state. This type of error, when it accumulates, can become disastrous because a large number of basic operation units are generally needed to perform a single computational task.

In digital computation, the process of error-correction is generally done at the logic gate level, to ensure that every logic gate outputs the correct binary result at high probability. In contrast, to illustrate the case of dynamical systems, we take the cubic bistable system (8) as an example. In the absence of noise, to obtain a desired output only requires diminishing energy: setting the initial state of the system to either $+\epsilon$ or $-\epsilon$ with arbitrarily small ϵ will be sufficient. However, noise is unavoidable in physical process. For instance, assume there is additive (but not necessarily Gaussian) noise represented by η, and so the system dynamics are

$$\dot{x} = f(x; \eta) = x(1 - x^2) + \eta. \tag{15}$$

Then, because of the presence of noise, the control u would need to change to ensure the system state is kept in the correct basin [35]. In fact, if noise is persistent, a single impulse control at the initial state is no longer sufficient. Instead, later-on control is generally needed in case noise kicks the state of the system to a different basin [36]. In this case, the control u would need to be larger and require more physical energy. In other cases, noise itself can even serve to regulate behavior of the system toward the desired output [37].

A more subtle question arises when considering a non-negligible effect of noise that might sometimes cause the state of the system to converge to the wrong stable state — this can happen, for instance, when $\eta(t)$ has large variance at the beginning which gradually decreases over time. In this case, one idea to correct the possible error is to follow the simple "repetition" trick, by evolving multiple copies of such systems, from different starting points. Depending on the characteristics of the dynamical system under consideration, two general strategies are possible. One possibility is to use completely independent systems followed by some aggregation (e.g. average or "majority vote") to collect the results of all and determine the final output. A second strategy is to use correlated dynamics, in fact adopting a network of coupled individual systems defining network dynamics as in Equation (11) with noise. Such an approach has proven successful for instance for accelerating the search of optimal solutions by employing correlated Monte Carlo simulations in methods known as *replica exchange MCMC* [38].

In general, under the dynamical systems framework of computation, we can treat error correction as a particular goal of control where additional control — beyond what is normally needed in the absence of "error" — is imposed to ensure robustness of the computation. For instance, in the optimal control equation (14), suppose that in the absence of disturbance we have $\lim_{t\to\infty} \boldsymbol{x}(t) = Q(\boldsymbol{x_0})$ but, because of various types of disturbance, this limit equation is violated. Then, the problem of error correction becomes one of designing the control signal $\boldsymbol{u}(t)$ so that the trajectory of the dynamics returns to its nominal path toward $Q(\boldsymbol{x}_0)$.

6 Discussion and Outlook

Modern computation has seen an explosive growth and its success forms a key element of the information age. Being able to precisely define and exercise a particular sequence of logic operations — a defining feature of digital computation — has dominated the thinking and industrial design of algorithms as well as computation hardware. After almost eighty years of progress since the birth of the first general-purpose digital computer ENIAC, the world surrounding us has witnessed significant changes and progress. It is not clear if the computational needs of the future could be addressed with current computational paradigms, or whether next-generation techniques of computation need to be developed. Indeed, current technological trends show an increasing shift toward special-purpose hardware, both digital hardware (e.g. FPGAs, GPUs, DPUs) and analog (e.g. quantum annealers), due to scaling limitations of current technologies. Moreover, it is becoming increasingly clear that, in many scenarios, large-scale computation does not necessarily require a completely precise, step-by-step accuracy. Instead, for many problems, only an end-to-end and sometimes a "good-enough" output would be acceptable. In these cases, alternative means of computation through physical systems could take advantage of these requirements to provide faster and more energy-efficient — albeit approximate — results. Representative directions include heuristic dynamics for large-scale optimization problems [39, 40].

To fully unlock the efficiency of physical systems for computation, a new type of framework and mathematical language is likely needed. Here we offer some viewpoints from dynamical systems and control theory together with a few lines of preliminary thoughts on what might be useful to define computation beyond Boolean. We hope that our perspectives, premature as they might be at this early stage, nevertheless serve to provide different ways to think of computation beyond mainstream approaches and inspire new ideas and research efforts that potentially revolutionize the theory and practice of computation.

References

[1] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[2] Alan M Turing. Intelligent machinery, a heretical theory. *The Turing test: Verbal behavior as the hallmark of intelligence*, 105, 1948.

[3] Charles H Bennett and Rolf Landauer. The fundamental physical limits of computation. *Scientific American*, 253(1):48–57, 1985.

[4] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[5] Igor L Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014.

[6] Claude E Shannon. Mathematical theory of the differential analyzer. *Journal of Mathematics and Physics*, 20(1-4):337–354, 1941.

[7] James D Meiss. *Differential dynamical systems*. SIAM, 2007.

[8] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[9] Kurt Bryan and Tanya L. Leise. The $25,000,000,000 eigenvector: The linear algebra behind google. *SIAM Rev.*, 48:569–581, 2006.

[10] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Phys. Rev. E*, 74:036104, Sep 2006.

[11] Hervé Abdi and Lynne J. Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.

[12] James W Demmel. *Applied numerical linear algebra*. SIAM, 1997.

[13] Amit Bhaya and Eugenius Kaszkurewicz. *Control Perspectives on Numerical Algorithms and Matrix Problems*. Society for Industrial and Applied Mathematics, 2006.

[14] L Bess. Hamiltonian dynamics and the schrödinger equation. *Progress of Theoretical Physics*, 52(1):313–328, 1974.

[15] Robert Alicki and Mark Fannes. Quantum dynamical systems. 2001.

[16] Tamás Vicsek and Anna Zafeiris. Collective motion. *Physics reports*, 517(3-4):71–140, 2012.

[17] Alexander N Pisarchik and Ulrike Feudel. Control of multistability. *Physics Reports*, 540(4):167–218, 2014.

[18] Mike Hurley. Attractors: persistence, and density of their basins. *Transactions of the American Mathematical Society*, 269(1):247–271, 1982.

[19] Edward Ott. *Chaos in dynamical systems*. Cambridge university press, 2002.

[20] Steven H Strogatz. *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering*. CRC press, 2018.

[21] Steven H Strogatz. From kuramoto to crawford: exploring the onset of synchronization in populations of coupled oscillators. *Physica D: Nonlinear Phenomena*, 143(1-4):1–20, 2000.

[22] Gyorgy Csaba and Wolfgang Porod. Coupled oscillators for computing: A review and perspective. *Applied physics reviews*, 7(1):011302, 2020.

[23] Jie Sun and Erik M Bollt. Causation entropy identifies indirect influences, dominance of neighbors and anticipatory couplings. *Physica D: Nonlinear Phenomena*, 267:49–57, 2014.

[24] Kunihiko Kaneko. Overview of coupled map lattices. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 2(3):279–282, 1992.

[25] Shawn D Pethel, Ned J Corron, and Erik Bollt. Symbolic dynamics of coupled map lattices. *Physical review letters*, 96(3):034105, 2006.

[26] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019.

[27] Michael S Branicky. Universal computation and other capabilities of hybrid and continuous dynamical systems. *Theoretical computer science*, 138(1):67–100, 1995.

[28] Hava T Siegelmann and Shmuel Fishman. Analog computation with dynamical systems. *Physica D: Nonlinear Phenomena*, 120(1-2):214–235, 1998.

[29] James P Crutchfield, William L Ditto, and Sudeshna Sinha. Introduction to focus issue: intrinsic and designed computation: information processing in dynamical systems—beyond the digital hegemony, 2010.

[30] David Sussillo. Neural circuits as computational dynamical systems. *Current opinion in neurobiology*, 25:156–163, 2014.

[31] Jie Sun and Adilson E Motter. Controllability transition and nonlocality in network control. *Physical review letters*, 110(20):208701, 2013.

[32] Rudolf Emil Kalman et al. Contributions to the theory of optimal control. *Bol. soc. mat. mexicana*, 5(2):102–119, 1960.

[33] Frank L Lewis, Draguna Vrabie, and Vassilis L Syrmos. *Optimal control*. John Wiley & Sons, 2012.

[34] Zhe Chen and Simon Haykin. On different facets of regularization theory. *Neural Computation*, 14(12):2791–2846, 2002.

[35] Adilson E Motter. Networkcontrology. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 25(9):097621, 2015.

[36] Daniel K Wells, William L Kath, and Adilson E Motter. Control of stochastic and induced switching in biophysical networks. *Physical Review X*, 5(3):031036, 2015.

[37] Rolf Landauer. Computation: A fundamental physical view. *Physica Scripta*, 35(1):88, 1987.

[38] David J Earl and Michael W Deem. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, 7(23):3910–3916, 2005.

[39] Alireza Marandi, Zhe Wang, Kenta Takata, Robert L Byer, and Yoshi- hisa Yamamoto. Network of time-multiplexed optical parametric oscillators as a coherent ising machine. *Nature Photonics*, 8(12):937–942, 2014.

[40] Hayato Goto. Bifurcation-based adiabatic quantum computation with a nonlinear oscillator network. *Scientific reports*, 6(1):1–8, 2016.

CHASPARK

Explore | Guide | Open | Enlighten

CHASPARK

HUAWEI TECHNOLOGIES CO., LTD.
Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129, PRC
Tel: +86-755-28780808